НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«На правах рукопису»
УДК 004.582

«До захисту допущено»
Науковий керівник кафедри
_____ І.А. Дичка
«___»_____2019 р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 121 Інженерія програмного забезпечення

на тему: «Модифікований спосіб кешування даних клієнтської бібліотеки Apollo-Client для GraphQL»

Виконав:
студент ІІ курсу, групи КП-81мп
Худер Карім Нідаль _____

Керівник:
Ст. викладач кафедри ПЗКС, к.т.н.,
Люшенко Л.А. _____

Консультант з нормоконтролю:
Доцент кафедри ПЗКС, к.т.н., доцент
Онай М.В. _____

Рецензент:
Доц. кафедри ММСА ІПСА, к.ф-м.н., доц.,
Шубенкова І.А. _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2019 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

Рівень вищої освіти – другий (магістерський) за освітньо-професійною програмою

Спеціальність (освітня програма) – 121 «Інженерія програмного забезпечення» ("Інженерія програмного забезпечення комп'ютерних та інформаційно-пошукових систем")

ЗАТВЕРДЖУЮ

Науковий керівник кафедри

_____ І.А. Дичка

«___»_____2018 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Худер Карім Нідаль

1. Тема дисертації «Модифікований спосіб кешування даних клієнтської бібліотеки Apollo-Client для GraphQL», науковий керівник дисертації Люшенко Леся Анатоліївна, к.т.н., затверджені наказом по університету від «13» листопада 2019 р. № 3895-С.

2. Термін подання студентом дисертації «16» грудня 2019 р.

3. Об'єкт дослідження: бібліотека Apollo Client, яка є найбільш популярним клієнтом GraphQL.

4. Предмет дослідження: механізм кешування даних в бібліотеці Apollo Client.

5. Перелік завдань, які потрібно розробити:
   - аналіз існуючого механізму кешування даних в бібліотеці Apollo Client;
   - аналіз існуючого API та основних архітектурних складових бібліотеки Apollo Client;
   - описати виявлено проблему кешування контекстно-залежних сутностей;
   - збереження існуючого API бібліотеки Apollo Client;
   - надання можливості створення власних правил обробки даних;
   - розробка Apollo Link Resolver, що оброблює результати GraphQL запитів з мережі;
   - розробка алгоритму модифікації даних на основі, вказаних розробником, правил;
   - підтримка усіх JavaScript середовищ;
   - розмір отриманої бібліотеки не має перевищувати 1 кілобайт;
   - 100-відсоткове покриття тексту програми функціональними тестами;
   - створення бізнес-моделі кінцевого продукту, що описує ключові моменти в організації діяльності, пов'язаної з поширенням розробленої бібліотеки для вирішення проблеми кешування контекстно-залежних сутностей в бібліотеці Apollo Client.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу:
   - загальна архітектура GraphQL;
   - загальна архітектура Apollo Client;

- схема модифікації мережевого шару;
- діаграма класів розробленої бібліотеки;
- алгоритм обробки даних;
- результати тестування.

7. Орієнтовний перелік публікацій:
   - Тези доповіді “ Платформа для створення шаблонних чат-магазинів в Telegram”

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Онай М.В., к.т.н., доцент		

9. Дата видачі завдання «25» жовтня 2018 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Ґрунтовне ознайомлення з предметною галуззю	17.11.2018	
2.	Визначення структури магістерської дисертації; вивчення літератури, пошук додаткової літератури, патентний пошук	04.12.2018	
3.	Робота над першим розділом магістерської дисертації; проведення наукового дослідження	15.02.2019	
4.	Проведення наукового дослідження; робота над другим розділом магістерської дисертації; розроблення програмного забезпечення	05.04.2019	
5.	Проведення наукового дослідження; робота над статтею за результатами наукового дослідження	15.05.2019	
6.	Проведення наукового дослідження; робота над третім розділом магістерської дисертації	15.06.2019	
7.	Завершення роботи над основною частиною магістерської дисертації; підготовка ілюстративного матеріалу; підготовка матеріалів доповіді на конференції ПМК-2019	13.11.2019	
8.	Оформлення текстової і графічної частини магістерської дисертації	05.12.2019	

Студент К.Н. Худер

Науковий керівник дисертації Л.А. Люшенко

РЕФЕРАТ

Актуальність теми. На сьогоднішній день REST став стандартом для формування API веб-додатків. Ця концепція десятиліттями використовувалася для створення веб додатків. Однак вона має певний набір недоліків, які стають дедалі критичнішими в процесі еволюції веб технологій. Розвиток JavaScript надав можливість створення складних систем на стороні клієнта. Однак такі системи накладають свої вимоги для роботи з сервером. Проблема перезавантаження даних, яка існує у REST, збільшує навантаження на мережу, не виконуючи корисної дії. Також проблема недостатнього завантаження даних змушує розробників створювати складні системи для простих дій. Все це призволе до створення абсолютно нового способу спілкування з сервером – GraphQL. Ця технологія вирішує більшість проблем усталеного REST, та дозволяє максимально ефективно використовувати мережу. За часом навколо GraphQL було сформовано цілу екосистему з продуктів, які забезпечують зручну роботу з ним. Технології були випробувані на тисячах проектах, та, наразі, використовуються у production багатьма компаніями світу. За останні декілька років був сформований основний кістяк бібліотек, які прийнято використовувати при розробці додатку з використанням технології GraphQL. Серед таких продуктів є бібліотека Apollo Client, яка є найпопулярнішим клієнтом для роботи з GraphQL. В ході аналізу роботи цієї бібліотеки було виявлено критичну проблему, яка, в деяких випадках, робить неможливим її використання. Проблема стосується роботи з контекстно-залежними сутностями. Враховуючи популярність бібліотеки, вирішення даної проблеми є актуальним завданням.

Об’єктом дослідження є механізм кешування даних в бібліотеці Apollo Client.

Предметом дослідження є алгоритми та методи кешування даних.

Мета роботи: розробка програмного забезпечення, яке вирішує проблему кешування контекстно-залежних сутностей у бібліотеці Apollo Client, не змінюючи її API.

Методи дослідження. В даній роботі використовуються методи теоретичного дослідження: аналіз, синтез та узагальнення. Також застосовувалися емпіричні методи: експеримент, спостереження, вимірювання та опис.

Наукова новизна роботи полягає в наступному:

1. Розроблено спосіб кешування даних для бібліотеки Apollo Client. Спосіб базується на модифікації ідентифікаторів сутностей до моменту кешування стандартним механізмом Apollo Client.
2. Розроблено загальний інтерфейс для правил обробки сутностей певних типів. Даний підхід дозволяє абстрагуватися від конкретної схеми даних. Шляхом створення правил, можливе формування унікальних ідентифікаторів для сутностей, що дозволяє вирішити проблему коректної роботи кешування контекстно-залежних об’єктів в бібліотеці Apollo Client.
3. Розроблено Apollo Link, який вбудовується в існуючий мережевий шар проекту та виконує модифікацію результатів запиту, які надходять з мережі.

Практична цінність отриманих в роботі результатів полягає в тому, що розроблену бібліотеку можливо використовувати в різних JavaScript середовищах для вирішення проблеми кешування контекстно-залежних сутностей в бібліотеці Apollo Client. Розроблена бібліотека не змінює існуюче API. Це дозволяє використовувати її, не замінуючи існуючу кодову базу проекту. Також, в залежності від потреб розробника, перед записом в кеш можливе форматування даних шляхом створення правил.

Апробація роботи. Основні положення і результати роботи були представлені та обговорювались на XII науковій конференції магістрантів

та аспірантів «Прикладна математика та комп'ютинг» ПМК-2019 (Київ, 13-15 листопада 2019 р.).

Структура та обсяг роботи. Магістерська дисертація складається з вступу, п'яти розділів, висновків та додатків.

У вступі надано загальну характеристику роботи, виконано оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень, сформульовано мету і задачі досліджень, показано наукову новизну отриманих результатів і практичну цінність роботи.

У першому розділі було проаналізовано основні проблеми технології клієнт-серверної взаємодії REST та основні складові частини специфікації GraphQL. Були проаналізовані основні складові бібліотеки Apollo Client та її архітектуру. Був проведений аналіз щодо формування мережевого шару та кешування. Виявлено, що підхід до кешування даних Apollo Client має багато переваг, серед яких: збереження даних у плоскому вигляді, швидкість та автоматичне оновлення запитів

У другому розділі було детально викладено рішення, що пропонується, а саме: модифікація процесу кешування в бібліотеці Apollo Client. Модифікації мережевого шару відбувається таким чином, що отримані дані з мережі оброблюються за вказаними розробником правилами. Передача контексту до відповідного правила в процесі обробки результату, надає можливість формування унікальних ідентифікаторів для сутностей.

У третьому розділі було описано архітектуру та основні частини розробленої бібліотеки, а саме: правила, що описують процес модифікації сутностей конкретних типів; реєстр правил та механізми роботи з ним; алгоритм обробки вхідних даних; Apollo Link Resolver, який вбудовується в існуючий мережевий шар та оброблює результати запитів з мережі.

У четвертому розділі було описано методологію тестування для розроблюваної системи, проаналізовано результати роботи та визначено

напрями для подальшого вдосконалення системи у майбутньому. За результати тестування формується звіт про те, що розроблена система не має помилок, які можуть призвести до виключних ситуацій при роботі. Також було досліджено розмір отриманої системи, який складає 779 байт у gzip версії.

У п’ятому розділі було сформовано та побудовано бізнес-модель стартапу для кінцевого продукту, що описує ключові моменти в організації комерційної діяльності, пов’язаної з поширенням розробленої бібліотеки для вирішення проблеми кешування контекстно-залежних сутностей в бібліотеці Apollo Client.

У висновках проаналізовано отримані результати роботи.

У додатках наведено загальну архітектуру GraphQL, архітектуру Apollo Client та Apollo Link, алгоритм модифікації даних, діаграму класів бібліотеки та схема модифікації мережевого шару.

Робота виконана на 71 аркуші, містить 2 додатки та посилання на список використаних літературних джерел з 29 найменувань. У роботі наведено 16 рисунків, 3 таблиці та 22 лістинги.

Ключові слова: кешування даних, Apollo Client, Apollo Link, GraphQL.

ABSTRACT

Actuality of theme. Today, REST has become the standard for web application API development. This concept has been used for decades to create web applications. However, it does have a number of drawbacks that are becoming more critical as web technologies evolve. JavaScript development has made it possible to create sophisticated client-side systems. However, such systems impose requirements for server operation. The problem of rebooting data that exists in REST increases the load on the network without performing a useful action. Also, the problem of underutilization forces developers to create sophisticated systems for simple action. All this will allow you to create a completely new way of communicating with the server - GraphQL. This technology solves most of the problems of established REST, and allows the most efficient use of the network. In time, a whole ecosystem of products was created around GraphQL to make it easy to use. Technologies have been tested on thousands of projects and are now being used in production by many companies in the world. Over the past few years, the main backbone of libraries has been formed, which is commonly used to develop an application using GraphQL technology. Such products include the Apollo Client Library, which is the most popular client for working with GraphQL. An analysis of the work of this library has identified a critical problem that, in some cases, makes it impossible to use. The problem concerns the work with context-dependent entities. Given the popularity of the library, solving this problem is an urgent task.

Object of research is a mechanism for caching data in the Apollo Client library.

Subject of research is algorithms and methods for caching data.

Research objective is to develop software that solves the problem of caching context-dependent entities in the Apollo Client library without changing its API.

Research methods. This paper uses methods of theoretical research: analysis, synthesis and generalization. Empirical methods were also used: experiment, observation, measurement and description.

Scientific novelty of the work is as follows:

1. A method of caching data for the Apollo Client library has been developed. The method is based on the modification of entity IDs until caching by the standard Apollo Client mechanism.
2. A common interface for rules for processing entities of certain types is developed. This approach allows you to abstract from a particular data schema. By creating rules, it is possible to generate unique entities for the entities, which can solve the problem of correct caching of context-dependent objects in the Apollo Client library.
3. Apollo Link has been developed that builds into the existing network layer of the project and modifies the query results that come from the network.

The practical value of the results obtained are that the developed library can be used in different JavaScript environments to solve the problem of caching context-dependent entities in the Apollo Client library. The developed library does not change the existing API. This allows it to be used without replacing the existing project codebase. Also, depending on the needs of the developer, formatting data by creating rules may be possible before writing to the cache.

Approbation. The main provisions and results of the work were presented and discussed at the XII Scientific Conference of Undergraduate and Graduate Students in Applied Mathematics and Computing PMK-2019 (Kyiv, November 13-15, 2019).

Structure and content of the thesis. The master's thesis consists of an introduction, five sections, conclusions and appendices.

The introduction provides a general description of the work, assesses the current state of the problem, substantiates the relevance of the research direction,

formulates the purpose and objectives of the research, shows the scientific novelty of the results obtained and the practical value of the work.

The first section analyzes the main issues of client-server REST technology and the main components of the GraphQL specification. The main components of the Apollo Client library and its architecture were analyzed. An analysis was performed on network layer formation and caching. The Apollo Client data caching approach has been found to have many benefits, such as flat data storage, speed, and automatic query updating

The second section a detailed solution was proposed, namely: modifying the caching process in the Apollo Client library. The modifications to the network layer occur in such a way that the received data from the network is processed according to the rules specified by the developer. Passing the context to the appropriate rule in the process of processing the result provides the ability to generate unique identifiers for the entities.

The third section describes the architecture and main parts of the developed library, namely: rules that describe the process of modifying entities of specific types; register of rules and mechanisms of work with it; algorithm of processing of input data; Apollo Link Resolver, which is built into an existing network layer and processes the results of requests from the network.

The fourth section describes the testing methodology for the system under development, analyzes the results and identifies directions for further improvement of the system in the future. According to the results of testing, a report is generated that the developed system has no errors that can lead to exceptional situations at work. The size of the resulting system, which is 779 bytes in the gzip version, was also investigated.

In the fifth section builds and builds a startup business model for the end product that describes key business organization issues related to the development of a developed library to address the problem of context-dependent entity caching in the Apollo Client library.

The conclusion is analyzed the results of this master's work.

The applications provide a general GraphQL architecture, an Apollo Client and Apollo Link architecture, a data modification algorithm, a library class diagram, and a network layer modification scheme.

The work is made on 71 sheets, contains 2 applications and links to the list of used literary sources of 29 titles. The paper presents 16 figures, 3 tables and 22 listings.

Keywords: data caching, Apollo Client, Apollo Link, GraphQL.

РЕФЕРАТ

Актуальность темы. На сегодняшний день REST стал стандартом для формирования API веб-приложений. Эта концепция десятилетиями использовалась для создания веб приложений. Однако она имеет определенный набор недостатков, которые становятся все критичнее в процессе эволюции веб технологий. Развитие JavaScript предоставил возможность создания сложных систем на стороне клиента. Однако такие системы накладывают свои требования для работы с сервером. Проблема перезагрузки данных, существует в REST, увеличивает нагрузку на сеть, не выполняя полезного действия. Также проблема недостаточной загрузки данных заставляет разработчиков создавать сложные системы для простых действий. Все это призволе к созданию совершенно нового способа общения с сервером - GraphQL. Эта технология решает большинство проблем устойчивого REST, и позволяет максимально эффективно использовать сеть. По времени вокруг GraphQL было сформировано целую экосистему из продуктов, которые обеспечивают удобную работу с ним. Технологии были опробованы на тысячах проектах, и, пока, используются в production многими компаниями мира. За последние несколько лет был сформирован основной костяк библиотек, которые принято использовать при разработке приложения с использованием технологии GraphQL. Среди таких продуктов является библиотека Apollo Client, которая является самым популярным клиентом для работы с GraphQL. В ходе анализа работы этой библиотеки было обнаружено критическую проблему, которая, в некоторых случаях, делает невозможным ее использование. Проблема касается работы с контекстно-зависимыми сущностями. Учитывая популярность библиотеки, решения данной проблемы является актуальной задачей.

Объектом исследования является механизм кэширования данных в библиотеке Apollo Client.

Предметом исследования являются алгоритмы и методы кэширования данных

Цель работы: разработка программного обеспечения, которое решает проблему кэширования контекстно-зависимых сущностей в библиотеке Apollo Client, не меняя ее API.

Методы исследования. В данной работе используются методы теоретического исследования: анализ, синтез и обобщение. Также применялись эмпирические методы: эксперимент, наблюдение, измерение и описание.

Научная новизна работы заключается в следующем:

1. Разработан способ кэширования данных для библиотеки Apollo Client. Способ базируется на модификации идентификаторов сущностей до момента кэширования стандартным механизмом Apollo Client.
2. Разработан общий интерфейс для правил обработки сущностей определенных типов. Данный подход позволяет абстрагироваться от конкретной схемы данных. Путем создания правил, возможно формирование уникальных идентификаторов для сущностей, позволяет решить проблему корректной работы кэширования контекстно-зависимых объектов в библиотеке Apollo Client.
3. Разработаны Apollo Link, который встраивается в существующий сетевой слой проекта и выполняет модификацию результатов запроса, которые поступают из сети.

Практическая ценность полученных в работе результатов заключается в том, что разработанную библиотеку можно использовать в различных JavaScript средах для решения проблемы кэширования контекстно-зависимых сущностей в библиотеке Apollo Client.

Разработанная библиотека не меняет существующее API. Это позволяет использовать ее, не меняя существующую кодовую базу проекта. Также, в зависимости от потребностей разработчика, перед записью в кэш возможно форматирование данных путем создания правил.

Апробация работы. Основные положения и результаты работы были представлены и обсуждались на ХИИ научной конференции магистрантов и аспирантов «Прикладная математика и компьютинг» ПМК-2019 (Киев, 13-15 ноября 2019).

Структура и объем работы. Магистерская диссертация состоит из введения, пяти глав, заключения и приложений.

Во введении дано общая характеристика работы, выполнена оценка современного состояния проблемы, обоснована актуальность направления исследований, сформулированы цели и задачи исследований, показано научную новизну полученных результатов и практическую ценность работы.

В первом разделе были проанализированы основные проблемы технологии клиент-серверного взаимодействия REST и основные составные части спецификации GraphQL. Были проанализированы основные составляющие библиотеки Apollo Client и ее архитектуру. Был проведен анализ по формированию сетевой слоя и кэширования. Выявлено, что подход к кэширования данных Apollo Client имеет много преимуществ, среди которых: сохранение данных в плоском виде, скорость и автоматическое обновление запросов

Во втором разделе детально изложены решения, предлагается, а именно: модификация процесса кэширования в библиотеке Apollo Client. Модификации сетевого слоя происходит таким образом, что полученные данные из сети обрабатываются по указанным разработчиком правилам. Передача контекста к соответствующему правила в процессе обработки

результата, предоставляет возможность формирования уникальных идентификаторов для сущностей.

В третьем разделе описано архитектуру и основные части разработанной библиотеки, а именно: правила, описывающие процесс модификации сущностей конкретных типов; реестр правил и механизмы работы с ним; алгоритм обработки входных данных; Apollo Link Resolver, который встраивается в существующий сетевой слой и обрабатывает результаты запросов из сети.

В четвертом разделе описано методологию тестирования для разрабатываемой системы, проанализированы результаты работы и определены направления для дальнейшего совершенствования системы в будущем. По результатам тестирования формируется отчет о том, что разработана система не имеет ошибок, которые могут привести к исключению при работе. Также были исследованы размер полученной системы, который составляет 779 байт в gzip версии.

В пятом разделе было сформировано и построено бизнес-модель стартапа для конечного продукта, описывает ключевые моменты в организации коммерческой деятельности, связанной с распространением разработанной библиотеки для решения проблемы кэширования контекстно-зависимых сущностей в библиотеке Apollo Client.

В выводах проанализированы полученные результаты работы.

В приложениях приведены общую архитектуру GraphQL, архитектуру Apollo Client и Apollo Link, алгоритм модификации данных, диаграмму классов библиотеки и схема модификации сетевого слоя.

Работа выполнена на 71 листе, содержит 2 приложения и ссылки на список использованных литературных источников из 29 наименований. В работе приведены 16 рисунков, 3 таблицы и 22 листинга.

Ключевые слова: кэширование данных, Apollo Client, Apollo Link, GraphQL.

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

API (Application Programming Interface) – це набір готових класів, процедур, функцій, структур і констант, що надаються додатком (бібліотекою, сервісом) для використання в зовнішніх програмних продуктах.

REST (Representational State Transfer, «передача репрезентативного стану») – підхід до архітектури мережевих протоколів, які забезпечують доступ до інформаційних ресурсів.

Фреймворк (framework) – інфраструктура програмних рішень, що полегшує розробку складних систем.

Токен (Token) – об'єкт, що створюється з лексеми в процесі лексичного аналізу.

Greenfield project – проект, що не має обмежень, накладених попередньою роботою.

React – JavaScript бібліотека з відкритим вихідним тестом програми для розробки користувацьких інтерфейсів.

Карта тексту програми (source map) – це JSON-файл, який містить інформацію про те, як транслювати текст програми до початкового вигляду.

ВСТУП

На сьогоднішній день REST став стандартом для формування API веб-додатків. Ця концепція десятиліттями використовувалася та має свої переваги та недоліки. Однак технології розвиваються, тому недоліки REST стають все більш критичними при сучасній розробці веб додатків.

Розвиток JavaScript надав можливість створення складних веб додатків. Однак такі системи накладають свої вимоги для роботи з сервером. Проблема перезавантаження даних, яка існує у REST, збільшує навантаження на мережу, не виконуючи корисної дії. Також проблема недостатнього завантаження даних змушує розробників створювати складні системи для простих дій.

Все це призвело до створення абсолютно нового способу спілкування з сервером – GraphQL [1]. Ця технологія вирішує більшість проблем усталеного REST, та дозволяє максимально ефективно використовувати мережу.

З часом навколо GraphQL було сформовано цілу екосистему з продуктів, які забезпечують зручну роботу з ним [2]. Технології були випробувані на тисячах проектах, та, наразі, використовуються у production багатьма компаніями світу.

За останні декілька років був сформований основний кістяк бібліотек, які прийнято використовувати при розробці додатку з використанням технології GraphQL. Серед таких продуктів є бібліотека Apollo Client, яка, на сьогоднішній день, я найпопулярнішим клієнтом для роботи з GraphQL.

Серед основних функцій, які виконує бібліотека Apollo Client є:

– валідація даних в залежності від схеми;

– формування запиту та його мініфікація;

– робота з мережею;

– кешування даних.

Тобто бібліотека виконує всю роботу з даними, а розробник створює бізнес логіку. Однак в ході детального аналізу роботи механізму кешування бібліотеки, було виявлено критичну помилку, яка, в деякий випадках, робить неможливим її використання. Проблема стосується роботи з контекстно-залежними сутностями. А саме, неможливість коректного кешування контекстно-залежних об’єктів.

Відповідно до цього, метою даного дослідження є розробка програмного забезпечення, яке вирішує описану проблему. Також важливим є збереження існуючого API бібліотеки Apollo Client, оскільки зміна API спричинить необхідність зміни існуючої кодової бази.

Таким чином, основною задачею даної магістерської роботи є створення та дослідження ефективності маловагомої бібліотеки для вирішення проблеми кешування контекстно-залежних сутностей у бібліотеці Apollo Client.

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ДЛЯ РОБОТИ З GRAPHQL

За останнє десятиліття REST став стандартом для розробки та проектування веб API. Однак ця концепція має певний набір недоліків, які стають дедалі критичнішими в процесі еволюції веб технологій [3].

Серед основних проблем слід виділити перезавантаження або завантаження зайвих даних. Перезавантаження означає, що клієнт завантажує більше інформації ніж потрібно в додатку. Якщо на екрані потрібно відображати список користувачів лише з їх іменами, то у API REST запит буде відбуватися на кінцеву точку /users та в результаті буде отримано JSON з даними користувача. Однак результатом є повна структура даних користувачів, яка може містити зайву, для даного прикладу, інформацію. Окрім незручності для розробника, зайва кількість даних накладає додаткове навантаження на мережу. Ця проблема є особливо актуальною при роботі з мобільним трафіком, оскільки з'являються обмеження по швидкості мережі та об'єму переданих даних.

Також значною проблемою є недостатнє завантаження даних, яку також називають проблемою $n + 1$. Недостатнє завантаження, зазвичай, означає, що конкретна кінцева точка не дає достатньої кількості необхідної інформації. Клієнту доведеться робити додаткові запити, щоб отримати необхідну структуру даних. Це може перерости в ситуацію, коли клієнту потрібно робити декілька конкретизованих послідовних запитів. В якості прикладу розглянемо додаток в якому також потрібно відобразити останніх трьох підписників на користувача. API надає додаткову кінцеву точку /users/<user-id>/followers. Для того, щоб мати змогу відобразити необхідну інформацію, додатку доведеться зробити один запит до кінцевої точки /users, а потім, для кожного з отриманих користувачів зробити, запит до кінцевої точки /users/<user-id>/followers.

1.1. Загальна ідея та аналіз технології GraphQL

GraphQL – це новий спосіб побудови API, який забезпечує більш ефективну, потужну та гнучку альтернативу REST. Загальна ідея GraphQL полягає в тому, що сервер надає єдину точку взаємодії з клієнтом та відповідає саме тими даними, які вимагає клієнт [4][5].

Основні складові GraphQL:

– Schema Definition Language (SDL) ;

– Query;

– Mutation;

– Realtime Updates with Subscriptions;

– Schema.

Schema Definition Language (SDL)

GraphQL має власну систему типів, яка використовується для визначення схеми API. Синтаксис для написання схеми називається Schema Definition Language.

Лістинг 1. SDL для визначення типу

```
type Person {
  name: String!
  age: Int!
  posts: [Post!]!
}
```

Query

Під час роботи з API REST дані завантажуються з конкретних кінцевих точок. Кожна кінцева точка має чітко визначену структуру інформації, яку вона повертає. Це означає, що вимоги до даних клієнта ефективно кодуються в URL-адресі до якої він підключається.

Підхід у GraphQL кардинально відрізняється. Замість того, щоб мати декілька кінцевих точок, які повертають фіксовану структуру даних, API GraphQL, як правило, надає єдину точку взаємодії з клієнтом. Структура повернених даних не є фіксованою і формується на основі запиту клієнта.

Такий підхід дозволяє гнучко налаштувати процес обміну даними між клієнтом та сервером.

Це означає, що клієнту необхідно надіслати більше інформації на сервер, щоб висловити свої потреби в даних - ця інформація називається запитом.

Лістинг 2. Приклад GraphQL Query

```
{
  allPersons {
    name
    posts {
      title
    }
  }
}
```

Mutation

Поряд із запитом інформації від сервера, більшості програм також потрібен певний спосіб внесення змін до даних, які зберігаються на сервері. З GraphQL ці зміни здійснюються за допомогою так званих мутацій.

Існує три типи мутацій:

– створення нових даних;

– оновлення наявних даних;

– видалення наявних даних.

Мутації мають таку ж синтаксичну структуру, що і запити, але завжди починаються з ключового слова мутації.

Лістинг 3. Приклад GraphQL mutation

```
mutation {
  createPerson(name: "Bob", age: 36) {
    name
    age
  }
}
```

Realtime Updates with Subscriptions

Ще одна важлива вимога для багатьох програм сьогодні - це з'єднання в режимі реального часу з сервером, щоб негайно отримувати інформацію про важливі події. Для цього GraphQL пропонує концепцію subscription.

Коли клієнт підписується на подію, він ініціює і підтримує стійке з'єднання з сервером. Щоразу, коли ця конкретна подія відбувається, сервер передає клієнту відповідні дані. На відміну від запитів та мутацій, які відповідають типовому циклу "запит-відповідь", підписки представляють собою потік даних, які надсилаються клієнту.

Підписки записуються з використанням того ж синтаксису, що і запити та мутації.

Лістинг 4. Приклад GraphQL subscription

```
subscription {
  newPerson {
    name
    age
  }
}
```

Schema

Схема є однією з найважливіших концепцій при роботі з API GraphQL. Вона визначає можливості API та те, як клієнти можуть взаємодіяти з ним. Schema часто розглядається як контракт між сервером і клієнтом.

Як правило, схема - це просто набір типів GraphQL. Однак при написанні схеми для API існують деякі особливі кореневі типи: Query, Mutation, Subscription.

Лістинг 5. Приклад GraphQL Schema

```
type Query {
  allPersons(last: Int): [Person!]!
}
type Mutation {
  createPerson(name: String!, age: Int!): Person!
}
```

Продовження лістингу 5

```
type Subscription {
  newPerson: Person!
}
type Person {
  name: String!
  age: Int!
  posts: [Post!]!
}
type Post {
  title: String!
  author: Person!
}
```

Було сформовано загальну архітектуру GraphQL та її складові частини.

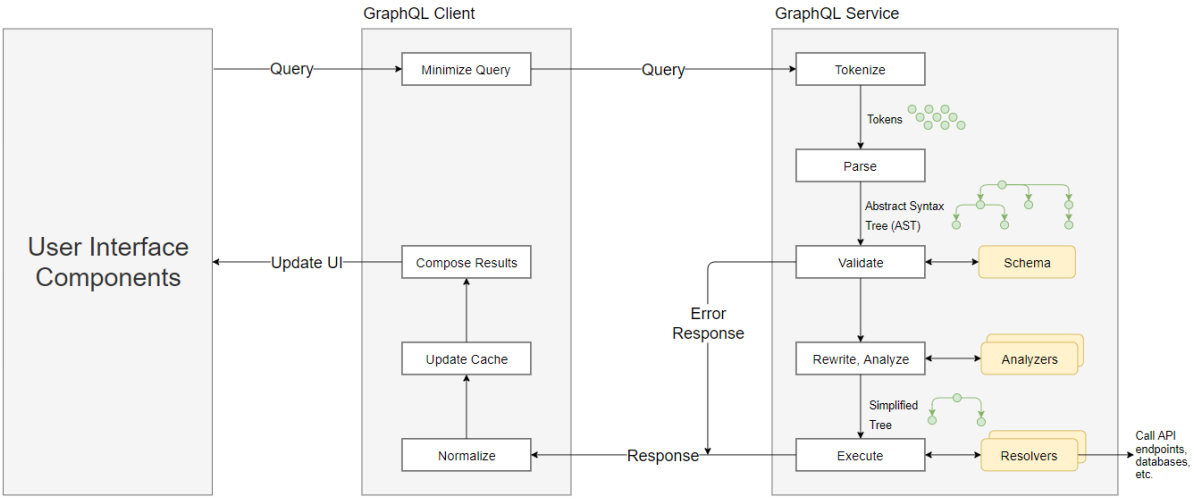


Рис. 1. Загальна архітектура GraphQL

На схемі видно, що клієнт взаємодіє лише з GraphQL Service. Query формується в компоненті інтерфейсу та передається на GraphQL Service за допомогою GraphQL Client. Розглянемо більш детально ці складові [6][7].

1.2. GraphQL client

GraphQL client не входить до специфікацій GraphQL, але значно спрощує процес управління даними. Різні клієнти по-різному оброблюють запити та відповіді.

Основні інфраструктурні функції GraphQL client:

– відправка GraphQL запитів на сервер;

– інтеграція з компонентами інтерфейсу;

– кешування даних;

– валідація та оптимізація GraphQL запитів на основі схеми.

На даний момент доступні два основні клієнти GraphQL. Перший - це Apollo Client, який спрямований спільнотою на створення потужного і гнучкого клієнта GraphQL для всіх основних платформ. Другий - Relay, і це домашній клієнт GraphQL Facebook, який значно оптимізує продуктивність і доступний лише для веб.

Було проаналізовано більш детально кожну з основних функцій GraphQL client.

Відправка GraphQL запитів на сервер

Головною перевагою GraphQL є те, що він дозволяє отримувати та оновлювати дані декларативно. Інакше кажучи, надає абстракцію більш високого рівня над API.

GraphQL client відповідає за конструювання, оптимізацію та відправку запиту через мережу. Компонента інтерфейсу лише вказує, які дані їй необхідні.

Інтеграція з компонентами інтерфейсу

GraphQL client після обробки отриманих даних відповідає за оновлення потрібної частини інтерфейсу. Залежно від платформ та каркасів, існують різні підходи до того, як відбувається процес оновлення інтерфейсу в цілому.

Для прикладу, GraphQL client для бібліотеки React використовує концепцію компонентів вищого порядку, щоб отримати необхідні дані, та зробити їх доступними у компоненті інтерфейсу.

Кешування даних

У більшості застосунків потрібно підтримувати кеш даних, які раніше були отримані з сервера. Наявність інформації на локальному рівні має важливе значення для забезпечення вільного користувальницького досвіду та зняття навантаження з серверу.

Кожен GraphQL client реалізує процес кешування по різному. Існують прості механізми, алгоритм яких полягає у збереженні хеш-таблиці, у якій ключем є запит, а значенням є відповідь з серверу. Такий спосіб збереження здатен виконувати свою функцію, але робить це не неефективно. Основна проблема при такому підході це дублювання даних, що призводить до швидкого збільшення об'єму кешу.

Більш вигідний підхід базується на попередній нормалізації даних. Це дозволяє отримати дані у пласкому вигляді та набір окремих записів з унікальними ідентифікаторами. За допомогою ідентифікаторів можливо швидко отримати та змінити дані.

Валідація та оптимізація GraphQL запитів на основі схеми

Оскільки схема містить всю інформацію про те, що клієнт потенційно може зробити з API, існує можливість перевіряти та оптимізувати запити, які клієнт може надіслати, в процесі збірки проекту.

GraphQL client може аналізувати усі запити GraphQL, які знаходиться в проекті, і порівнювати їх з інформацією зі схеми. Це дозволяє відловлювати більшу частину помилок на етапі розробки продукту.

1.3. GraphQL service

Як видно з рис. 1, GraphQL Server надає єдину кінцеву точку для взаємодії з клієнтом. GraphQL service повинен розпізнати та зрозуміти GraphQL запит, перш ніж він зможе відповісти правильними даними.

GraphQL service токенізує рядок запиту, а потім аналізує отриманий набір токенів для створення абстрактного синтаксичного дерева (AST). Потім відбувається процес валідації з існуючою схемою. В разі помилки,

відповідь про неї повертається клієнтові. Якщо запит валідний, AST може бути зменшений до форми, простішої у виконанні. Якщо є визначені аналізатори запитів, вони будуть викликані. Під час виконання викликаються усі resolver для отримання фактичних даних, що стосуються кожного поля. Функція resolver використовується для отримання даних для відповідного поля. В цій функції відбувається запит до бази даних, або до стороннього сервера. Після повернення результатів усіх resolver, GraphQL сервер буде пакувати дані у форматі, який був описаний запитом, і відправляти їх назад клієнту.

Було проаналізовано 3 архітектури, які включають GraphQL Service:

– GraphQL server з підключеною базою даних;
– GraphQL server, який є тонким шаром перед низкою сторонніх або застарілих систем та інтегрує їх через єдиний GraphQL API;
– гібридний підхід, при якому GraphQL server має власну базу даних, та інтегрується з сторонніми системами, формуючи єдиний GraphQL API.

GraphQL server з підключеною базою даних

Ця архітектура є найпоширенішою для greenfield projects. При такій архітектурі існує один веб сервер, який реалізує специфікацію GraphQL. Коли запит надходить на сервер GraphQL, сервер зчитує корисне навантаження запиту і отримує необхідну інформацію з бази даних. Наступним кроком формується об'єкт відповіді та повертається клієнтові.

GraphQL є незалежним від транспортного рівня. Це означає, що, потенційно, його можна використовувати з будь-яким доступним мережевим протоколом. Отже, можлива реалізація сервера GraphQL на основі HTTP, TCP, WebSockets тощо.

GraphQL також є незалежним від типу бази даних або формату, який використовується для зберігання даних. Тому можлива інтеграція GraphQL server з такими базами даних як MySQL, PostgreSQL, MongoDB тощо.

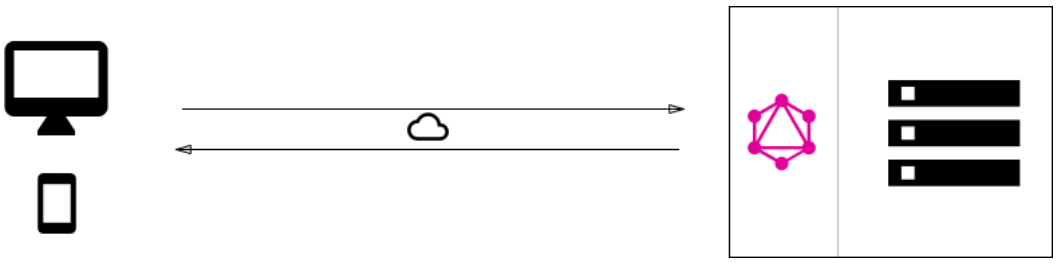


Рис. 2. GraphQL server з підключеною базою даних

GraphQL server який інтегрує існуючі системи

При цій архітектурі GraphQL server виконує інтеграцію декількох існуючих систем в єдине узгоджене API. Така архітектура актуальна при роботі з великою кількістю сервісів, які мають своє API.

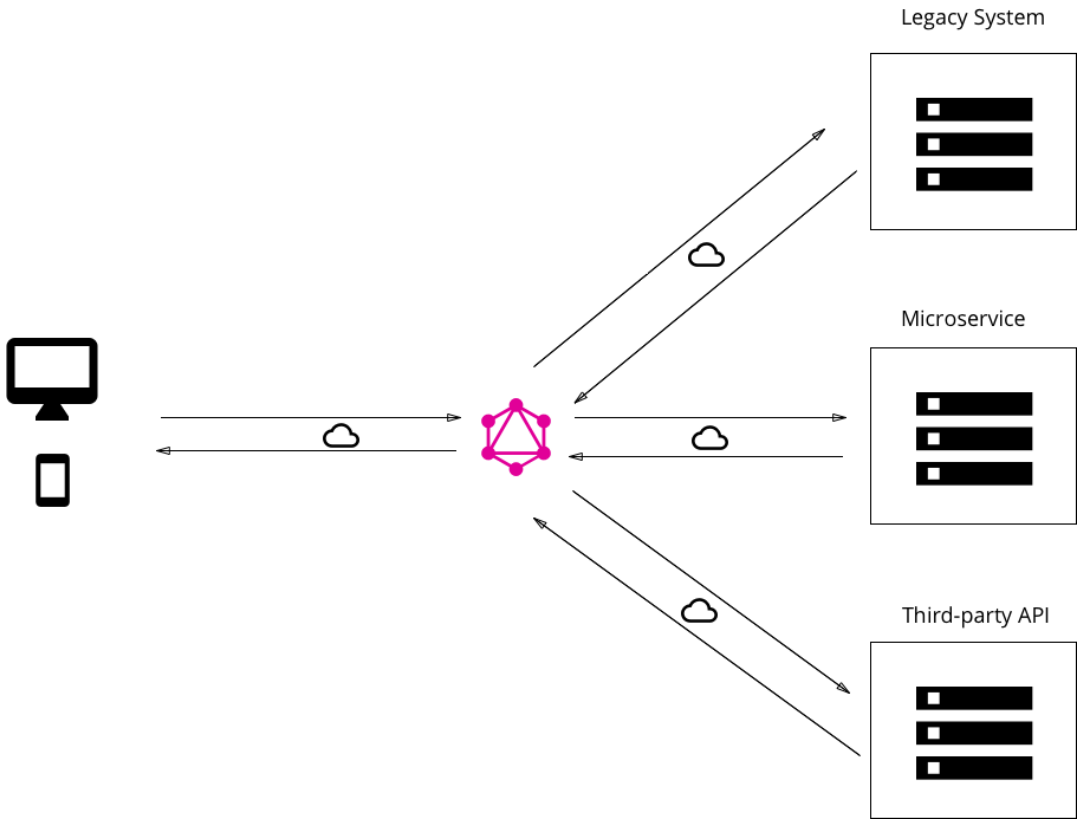


Рис. 1. GraphQL server інтегрує стороні системи у єдине API

Гібридний підхід

Поєднавши два попередні підходи можна побудувати GraphQL server, який має підключену базу даних та спілкується з сторонніми ресурсами.

Коли сервер отримує запит, за допомогою функцій resolver приймається рішення щодо ресурсу, який надає дані.

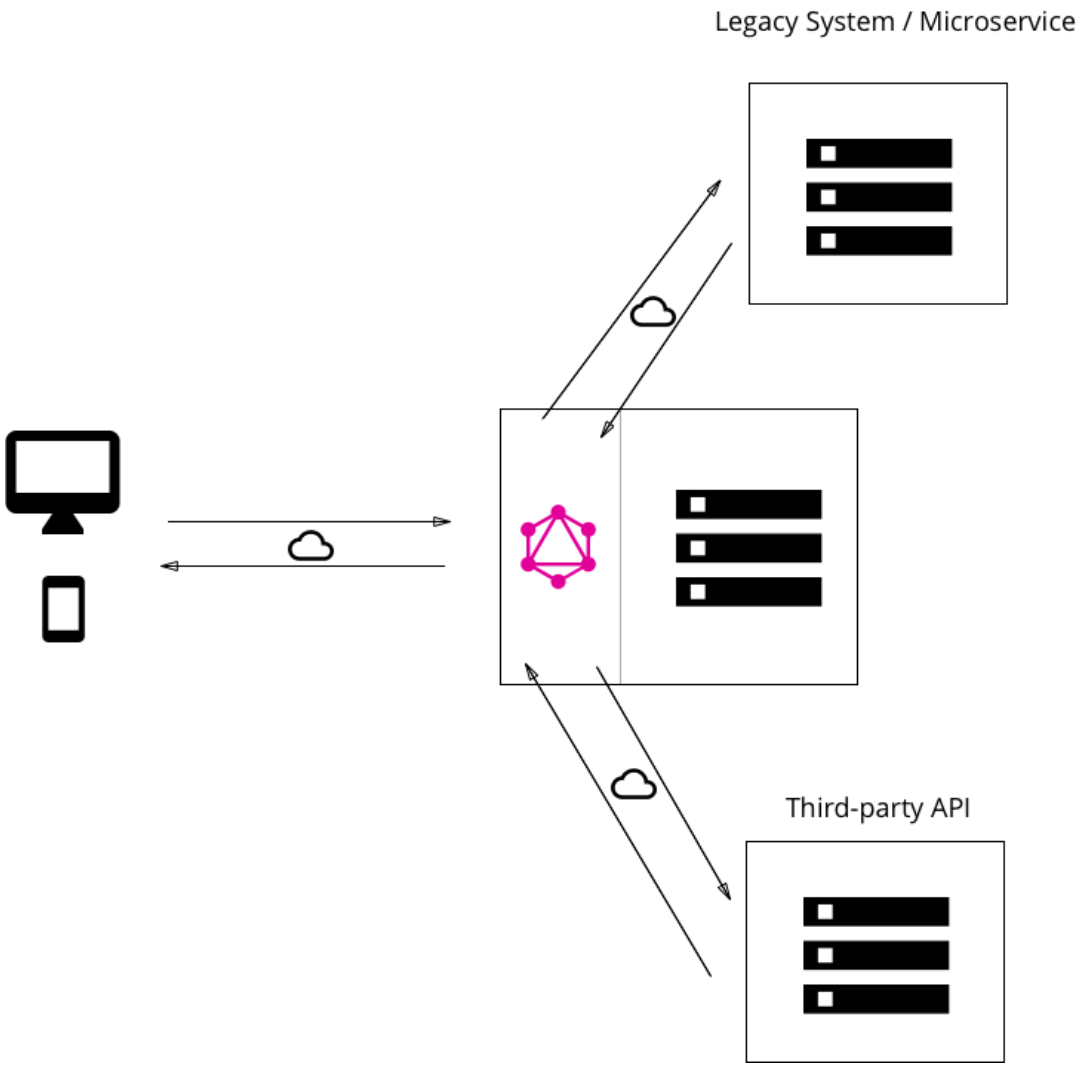


Рис. 2. Гібридний підхід до побудови архітектури з GraphQL server

1.4. Аналіз Apollo Client

Apollo Client – найпопулярніший клієнт для GraphQL. Apollo Client імплементує всі інфраструктурні функції, які були розглянуті в пункті 1.1.1.

Збір даних за допомогою Apollo Client дозволяє структурувати та писати текст програми декларативним способом, що відповідає сучасним найкращим практикам.

Apollo client підтримує велику кількість платформ:

– JavaScript
  - Angular
  - Vue
  - Meteor
  - Ember

– Web Components
  - Apollo Elements

– Native mobile
  - Native iOS with Swift
  - Native Android with Java

Розглянемо загальну архітектуру Apollo client [9].

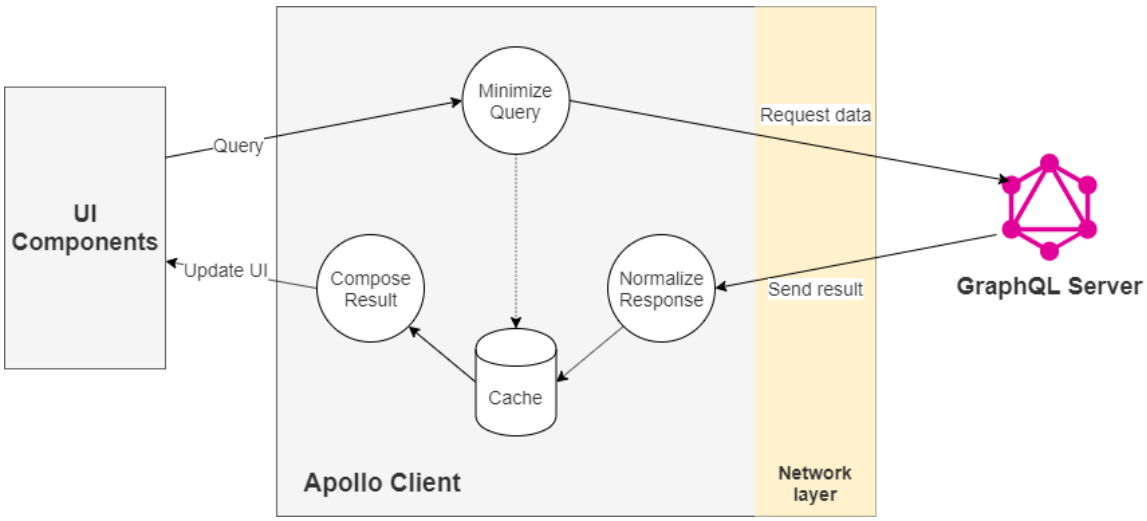


Рис. 3. Архітектура Apollo client

До основних складових Apollo Client належать: робота з кешем та мережевий шар. Apollo client мініфікує запит від компоненти інтерфейсу, та, в залежності від політики роботи з кешем, передає запит до мережевого шару або вичитує з кешу.

Apollo client пропонує декілька політик роботи з кешем (fetch policy):

– cache-first: політика за замовчуванням, коли ми завжди спершу намагаємося прочитати дані з локального кешу. Якщо всі дані,

необхідні для виконання запиту, знаходяться в кеші, ці дані будуть повернуті. Apollo client зробить запит на сервер тільки в разі відсутності необхідних даних у локальному кеші. Ця політика щодо отримання даних має на меті мінімізувати кількість мережевих запитів, надісланих від компоненти інтерфейсу.

– cache-and-network: Apollo client спершу намагатиметься прочитати дані з кешу. Якщо всі дані, необхідні для виконання запиту, знаходяться в кеші, ці дані будуть повернуті. Однак, запит завжди буде виконаний за допомогою мережевого інтерфейсу. Ця політика дозволяє користувачеві отримати швидку відповідь, а також підтримувати кешовані дані у відповідності з вашими даними сервера за рахунок додаткових мережевих запитів.
– network-only: при такій політиці початкові дані не будуть вичитуватися з кешу. Натомість Apollo client завжди буде робити запит, використовуючи мережевий інтерфейс, до сервера. Ця політика щодо отримання даних вирішує проблему узгодженості даних, але ціною миттєвої відповіді користувачеві.
– cache-only: ця політика отримання даних ніколи не виконуватиме запит, використовуючи мережевий інтерфейс. Натомість Apollo client завжди намагатиметься читати з кеша. Якщо даних запиту в кеші немає, буде видана помилка. Ця політика щодо отримання даних дозволяє взаємодіяти лише з даними у кеш-пам'яті локального клієнта, не роблячи жодних мережевих запитів.
– no-cache: ця політика отримання даних ніколи не поверне початкові дані з кешу. Apollo client завжди буде робити запит, використовуючи мережевий інтерфейс до сервера. На відміну від політики network-only отримані дані не будуть записані до кешу.

Було проаналізовано архітектуру та основні складові мережевого шару. Основним компонентом для побудови мережевого шару в Apollo Client є Apollo Link [10].

Рис. 4. Apollo Link

Apollo Link розроблений як потужний спосіб компонування дій навколо обробки даних за допомогою GraphQL. Кожне посилання являє собою підмножину функціональних можливостей, яку можна об’єднати з іншими посиланнями для створення складних потоків управління даними.

На базовому рівні Apollo Link – це функція, яка отримує операцію і повертає Observable. Операція – це об'єкт із наступною інформацією:

– query: DocumentNode, що описує операцію;

– variables: об’єкт зі змінними, що надсилаються під час операції;

– operationName: ім'я рядка запиту;

– extensions: об’єкт з даними про розширення;

– getContext: функція повернення контексту запиту;

– setContext: функція, яка приймає або новий об'єкт нового контексту, або функцію, яка отримує попередній контекст і повертає новий;

– toKey: функція для перетворення поточної операції в рядок, який буде використовуватися як унікальний ідентифікатор.

Алгоритм роботи мережевого шару полягає в послідовній передачі запиту до кожного Apollo Link один за одним.

Рис. 5. Мережевий шар Apollo Client

В основі Apollo Link лежить метод request. Він приймає наступні аргументи:

– operation: операція, що передається через Apollo Link;

– forward: (необов'язково) вказує на наступний Apollo Link в ланцюзі.

Метод request Apollo Link викликається для кожної операції, що проходить через ланцюг зв'язку.

Оскільки ланцюг має закінчуватися отриманням даних, Apollo Link мають поняття:

– non-termination link: Apollo Link, який використовує аргумент forward, тим самим продовжуючи ланцюг;

– termination link: Apollo Link, завданням якого є перетворення операції на результат. Зазвичай взаємодії з сервером та являє собою кінцеву ланку ланцюга.

Apollo Link можна використовувати як middleware для виконання побічних ефектів, модифікації операції, логування вхідних запитів. А також в якості afterware для виконання обробки результату операції та помилок.

Композиція ланцюгів формується двома основними способами:

– добавка (additive): об’єднання декількох Apollo Link у один ланцюг;

– спрямованість (directional): використання різних Apollo Link в залежності від операції.

1.5. Аналіз способу кешування даних бібліотекою Apollo client

Apollo Client використовує нормалізацію даних перед збереженням в кеш. Основна перевага такого рішення полягає в тому, що дані зберігаються у плоскому вигляді. Було проаналізовано основні кроки алгоритму нормалізації даних, який використовується в Apollo Client [8]:

– розбиття результатів на окремі об’єкти;

– присвоєння унікальних ідентифікаторів для кожного об’єкту;

– збереження у плоскому вигляді.

Збереження даних у плоскому вигляді має наступні переваги:

– Швидкість. Отримання даних по ключу з такого кешу виконується за час O(1).
– Розмір. Якщо два запити повертають об’єкт з однаковим ідентифікатором, то в кеш буде збережений один об’єкт, а в результат запитів буде присвоєно посилання на закешований об’єкт.

Було проаналізовано механізми, які використовує Apollo client для призначення унікальних ідентифікаторів [11].

За замовченням Apollo client намагається генерувати унікальний ідентифікатор об’єкта, поєднуючи поле __typename об'єкта з його полем id або _id. Якщо об’єкт не містить поля __typename або поля id (_id), Apollo client приймає шлях до об’єкту в контексті запиту в якості унікального ідентифікатора (наприклад, ROOT_QUERY.books.0 для першого запису, повернутого для кореневого запиту books). В документації Apollo Client зазначено, що дана стратегія є небажаною, оскільки створює кешовані дані під окремі запити. Це означає, що якщо кілька запитів повертають однаковий об’єкт, кожен запит неефективно кешує окремий екземпляр цього об’єкта.

Apollo client дозволяє створити власну стратегію генерування унікального ідентифікатора для об’єктів. Налаштування відбувається шляхом передачі функції генерування в поле dataIdFromObject до конструктора Apollo cache. Функція приймає об’єкт даних і повертає унікальний ідентифікатор, який буде використаний при нормалізації даних у кеш.

Було проаналізовано роботу Apollo Cache на прикладі GraphQL запиту, який наведений у лістингу 6.

Лістинг 6. GraphQL query для аналізу кешування

```
{
  books {
    title
    author {
      id
      name
    }
  }
}
```

Для більш зручного візуального аналізу дані представлені у виді граф замість JSON [8].

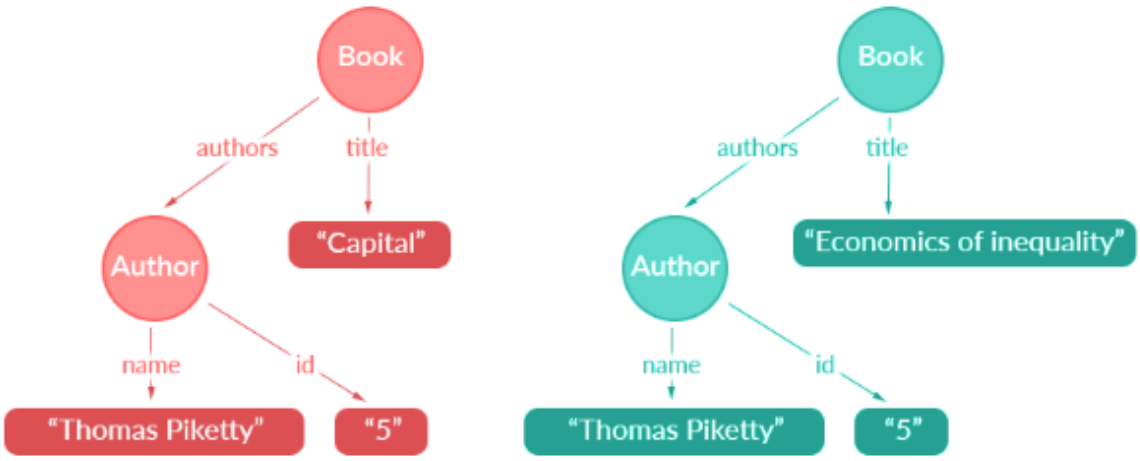


Рис. 6. Результат запиту books у вигляді графу

Як видно з рис. 8 в результаті було отримано дві книги, які були видані одним автором. За замовченням об'єкту автору буде присвоєний унікальний ідентифікатор: Author:5. Після нормалізації отриманих даних в кеш буде збережена наступна структура.

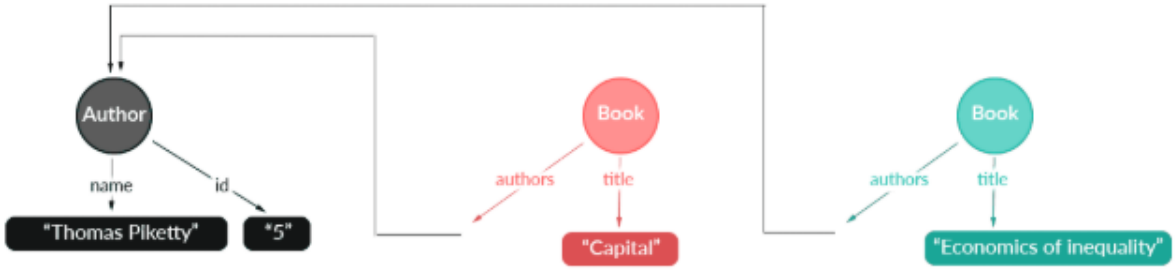


Рис. 7. Нормалізований результат запиту books у вигляді графу

Даний підхід до формування кешу в Apollo client дозволяє автоматично оновлювати дані. Якщо будь-який із переглянутих вузлів дерева запитів змінить значення, запит буде оновлено з новим результатом.

Лістинг 7. GraphQL mutation для редагування автора

```
mutation {
  updateAuthor(id: "5" name: "Adam Smith") {
    id
    name
  }
}
```

В результаті виконання такого запиту буде повернуто модифікований об’єкт автора з унікальний ідентифікатором: Author:5. Оскільки кеш містить запис з таким ідентифікатором, дані в кеші будуть оновлені отриманими даними. Запити, які містять посилання на автора з цим ідентифікатором будуть оновлені.

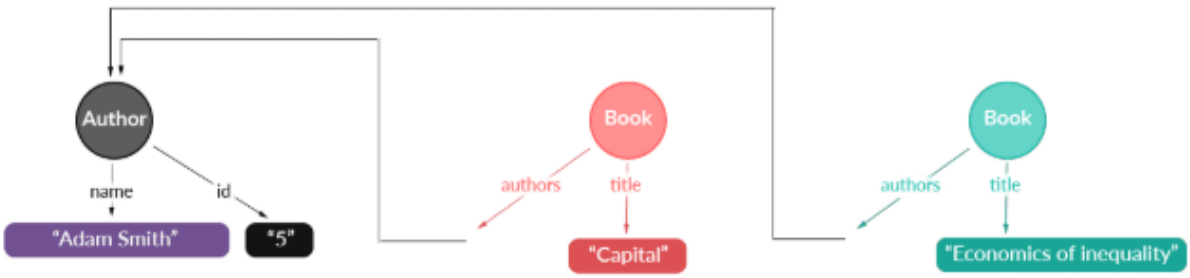


Рис. 8. Структура кешу після оновлення автора

1.6. Висновки до розділу 1

В даному розділі було проаналізовано основні проблеми технології клієнт-серверної взаємодії REST, серед яких: перезавантаження та недостатнє завантаження даних.

Досліджено та сформовано ряд переваг технології GraphQL:

- добре підходить для складних систем та мікросервісної архітектури [12];
- єдина точка взаємодії з клієнтом;
- вирішена проблема перезавантаження та недостатнього завантаження даних;
- формування даних в залежності від клієнтського запиту;
- валідація даних;
- автоматичне генерування документації для API.

В даному розділі проаналізовано основні складові частини специфікації GraphQL та розглянуті способи побудови GraphQL Server та його взаємодії з GraphQL Client.

Проаналізовані основні складові бібліотеки Apollo Client та її архітектура. Був проведений аналіз щодо формування мережевого шару та кешування. Виявлено, що підхід до кешування даних Apollo Client має багато переваг, серед яких: збереження даних у плоскому вигляді, швидкість та автоматичне оновлення запитів.

2. МОДИФІКОВАНИЙ СПОСІБ КЕШУВАННЯ ДАННИХ

В ході аналізу найпопулярнішого клієнту для GraphQL Apollo Client, було сформованого переваги та недоліки існуючого способу кешування.

Серед основних переваг існуючого методу є спосіб збереження даних у пласкому вигляді. Такий підхід запобігає дублюванню об’єктів, та дозволяє автоматично оновлювати результати запитів. Недоліком такого підходу є неможливість коректної роботи з контекстно-залежними об’єктами.

2.1. Проблема нормалізації даних в бібліотеці Apollo Client

Провівши аналіз механізму кешування Apollo Client було виявлено проблему, яка, в деяких випадках, робить неможливим його використання. Проблема полягає в контекстних ідентифікаторах. Оскільки кеш в Apollo client зберігається у пласкому вигляді, об’єкти з однаковим ідентифікатором перетирають один одного. Це зручно для автооновлення та запобігає збереженню зайвих екземплярів одного об’єкту. Але в той час робить неможливим його використання при збереженні контекстних об’єктів.

Контекстний ідентифікатор – ідентифікатор, який є унікальним в рамках свого контексту.

Для прикладу, розглянемо структуру даних з лістингу 8.

Лістинг 8. GraphQL результат

```
{
  "sportEvent": {
    "id": "sp1",
    "__typename": "SportEvent",
    "markets": [
      {
        "id": "m1",
        "title": "winner",
        "__typename": "Market",
        "odds": [
          {
            "id": "1",
            "value": 1.5,
            "__typename": "Odd"
          },
          {
```

Продовження лістингу 8

```
            "id": "2",
            "value": 1.75,
            "__typename": "Odd"
          }
        ]
      },
      {
        "id": "m2",
        "title": "First blood",
        "__typename": "Market",
        "odds": [
          {
            "id": "1",
            "value": 1.1,
            "__typename": "Odd"
          },
          {
            "id": "2",
            "value": 2.05,
            "__typename": "Odd"
          }
        ]
      }
    ]
  }
}
```

Результат містить сутності трьох типів: SportEvent, Market, Odd. Слід зазначити, що поле id сутності Market унікальне тільки в рамках конкретної сутності SportEvent. Також, поле id сутності Odd унікальне тільки в рамках свого Market.

В рамках стандартного механізму нормалізації даних Apollo client, отримаємо результат, який наведено у лістингу 9.

Лістинг 9. Нормалізована відповідь в рамках стандартного механізму

```
{
  "SportEvent:sp1": {
    "id": "sp1",
    "__typename": "SportEvent",
    "markets": [Market:m1, Market:m2] // посилання на об’єкти
  },
  "Market:m1": {
    "id": "m1",
    "title": "winner",
    "__typename": "Market",
    "odds": [Odd:1, Odd:2] // посилання на об’єкти
  },
  "Market:m2": {
```

Продовження лістингу 9

```
    "id": "m2",
    "__typename": "Market",
    "title": "First blood",
    "odds": [Odd:1, Odd:2] // посилання на об’єкти
  },
  "Odd:1": {
    "id": "1",
    "value": 1.1,
    "__typename": "Odd"
  },
  "Odd:2": {
    "id": "2",
    "value": 2.05,
    "__typename": "Odd"
  }
}
```

В результаті такої нормалізації було втрачено дані Odd для Market:m1. Тепер перший і другий маркет мають посилання на однакові Odd. До такого результату призводить стандартний механізм присвоєння унікальних ідентифікаторів для кожної сутності в процесі нормалізації даних Apollo Client.

Для Apollo Client важливо, щоб ідентифікатор об’єкту був унікальним в рамках всієї системи.

Apollo Client не рекомендує, але дозволяє, відключити стандартний спосіб визначення ідентифікаторів, шляхом вказання стратегії вибору ідентифікатора.

Лістинг 10. Відключення стандартного способу визначення ідентифікатора

```
dataIdFromObject: object => {
  if(
    object.__typename === 'Market' ||
    object.__typename === 'Odd'
  ) return null
}
```

В такому випадку, ідентифікатор буде формуватися на основі шляху до об’єкту і буде отримано результат, який наведено у лістингу 11.

Лістинг 11. Нормалізована відповідь з відключеним стандартним способом визначення ідентифікатора

```
{
  "SportEvent:sp1": {
    "id": "sp1",
    "__typename": "SportEvent",
    "markets":  [SportEvent:sp1.markets.0,  SportEvent:sp1.markets.1]  //
посилання на об’єкти
  },
  "SportEvent:sp1.markets.0": {
    "id": "m1",
    "title": "winner",
    "__typename": "Market",
    "odds":[SportEvent:sp1.markets.0.odds.0,
SportEvent:sp1.markets.0.odds.1] // посилання на об’єкти
  },
  "SportEvent:sp1.markets.1": {
    "id": "m2",
    "__typename": "Market",
    "title": "First blood",
    "odds":[SportEvent:sp1.markets.1.odds.0,
SportEvent:sp1.markets.1.odds.1] // посилання на об’єкти
  },
  "SportEvent:sp1.markets.0.odds.0": {
    "id": "1",
    "value": 1.5,
    "__typename": "Odd"
  },
  "SportEvent:sp1.markets.0.odds.1": {
    "id": "2",
    "value": 1.75,
    "__typename": "Odd"
  },
  "SportEvent:sp1.markets.1.odds.0": {
    "id": "1",
    "value": 1.1,
    "__typename": "Odd"
  },
  "SportEvent:sp1.markets.1.odds.1": {
    "id": "2",
    "value": 2.05,
    "__typename": "Odd"
  }
}
```

В такому випадку, всі дані були збережені правильно, але тепер ідентифікатори сутностей Odd і Market залежать від порядкового номеру в масиві. Це вносить незручності у роботі з кешем, а саме в процес зчитування та внесення змін. Також, це може призвести до помилок, які важко відстежити. Для прикладу, отримаємо по subscription дані, які наведено у лістингу 12.

Лістинг 12. GraphQL результат subscription onChangeSportEvent

```
{
  "sportEvent": {
    "id": "sp1",
    "__typename": "SportEvent",
    "markets": [
      {
        "id": "m2",
        "title": "First blood",
        "__typename": "Market",
        "odds": [
          {
            "id": "1",
            "value": 5,
            "__typename": "Odd"
          },
          {
            "id": "2",
            "value": 10,
            "__typename": "Odd"
          }
        ]
      }
    ]
  }
}
```

```
{
  "SportEvent:sp1": {
    "id": "sp1",
    "__typename": "SportEvent",
    "markets": [
      SportEvent:sp1.markets.0
    ]
  },
  "SportEvent:sp1.markets.0": {
    "id": "m2",
    "__typename": "Market",
    "title": "First blood",
    "odds": [
      SportEvent:sp1.markets.0.odds.0,
      SportEvent:sp1.markets.0.odds.1
    ]
  },
  "SportEvent:sp1.markets.0.odds.0": {
    "id": "1",
    "value": 5,
    "__typename": "Odd"
  },
  "SportEvent:sp1.markets.0.odds.1": {
    "id": "2",
    "value": 10,
    "__typename": "Odd"
  }
}
```

В підписці отримаємо патч інформацію, тобто тільки ті Market, які були змінені на сервері. Зліва представлено оригінальну відповідь з сервера, а з правої частини її нормалізований вид. Був змінений Market з id m2, але через те, що тепер в масиві markets він знаходиться на першому місці, Apollo Client присвоює йому вже існуючий ідентифікатор: SportEvent:sp1.markets.0, тим самим перезаписавши не той об’єкт. В результаті накладання цих даних на існуючий кеш отримаємо дані, які наведено у лістингу 13.

Лістинг 13. Нормалізовані дані після отримання патч інформації

```
{
  "SportEvent:sp1": {
    "id": "sp1",
    "__typename": "SportEvent",
    "markets": [SportEvent:sp1.markets.0, SportEvent:sp1.markets.1]   //
посилання на об’єкти
  },
  "SportEvent:sp1.markets.0": {
    "id": "m2",
    "title": "First blood",
```

Продовження лістингу 13

```
    "__typename": "Market",
    "odds":[SportEvent:sp1.markets.0.odds.0,
SportEvent:sp1.markets.0.odds.1]  // посилання на об’єкти
  },
  "SportEvent:sp1.markets.1": {
    "id": "m2",
    "title": "First blood",
    "__typename": "Market",
    "odds":[SportEvent:sp1.markets.1.odds.0,
SportEvent:sp1.markets.1.odds.1] // посилання на об’єкти
  },
  "SportEvent:sp1.markets.0.odds.0": {
    "id": "1",
    "value": 5,
    "__typename": "Odd"
  },
  "SportEvent:sp1.markets.0.odds.1": {
    "id": "2",
    "value": 10,
    "__typename": "Odd"
  },
  "SportEvent:sp1.markets.1.odds.0": {
    "id": "1",
    "value": 1.1,
    "__typename": "Odd"
  },
  "SportEvent:sp1.markets.1.odds.1": {
    "id": "2",
    "value": 2.05,
    "__typename": "Odd"
  }
}
```

Проаналізувавши усі механізми нормалізації даних Apollo Client було сформовано звіт про неможливість коректної роботи з контекстними ідентифікаторами.

2.2. Модифікований спосіб кешування даних

Після аналізу усіх переваг існуючого механізму було прийнято рішення не вносити зміни до існуючого API роботи з кешем. Це дає змогу використовувати запропонований спосіб не вносячи зміни до існуючої кодової бази.

Запропонований спосіб базується на модифікації мережевого шару бібліотеки Apollo Client. У підрозділі 1.4 було проаналізовано роботу мережевого шару та способи його модифікації.

Рішенням проблеми кешування контекстно-залежних об’єктів є створення додаткового Apollo Link, який буде оброблювати ідентифікатори об’єктів за певними правилами та має назву Apollo link resolver.

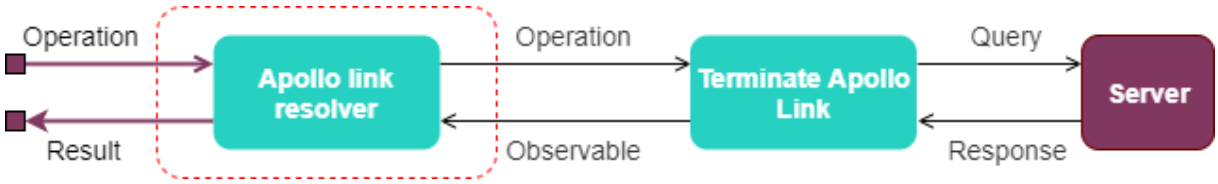


Рис. 9. Модифікація мережевого шару

На рис. 11 видно, що розроблений Apollo Link має бути розташований перед termination link. При даному способі Apollo Link resolver використовується в якості afterware.

Послідовність дій, яку виконує запропонований Apollo link resolver:

1. Отримання Operation від попереднього Apollo Link.
2. Виконання наступного Apollo Link та підпис на його результат.
3. Отримання результату.
4. Модифікація отриманого результату.
5. Повернення модифікованого результату.

Модифікація результату відбувається за правилами, які отримує Apollo link resolver при створенні.

Використання такого підходу разом з існуючою системою кешування дозволяє працювати з контекстно-залежними об’єктами, оскільки з’являється можливість модифікувати ідентифікатори.

Для коректної роботи існуючої системи кешування Apollo Client потрібно, щоб усі ідентифікатори були глобально-унікальними. В такому разі, при нормалізації та створені плоскої структури, дані не будуть перетиратися.

Використовуючи запропонований Apollo link resolver можна модифікувати ідентифікатори в отриманих об’єктах до того, як вони потраплять до етапу нормалізації. Склеївши власний ідентифікатор з

ідентифікатором контекстного об’єкту, отримаємо абсолютно унікальний ідентифікатор в рамках всього додатку. Таким чином, виконується основна вимога Apollo Client до кешування, яка полягає у наявності глобально-унікальних ідентифікаторах.

Використавши Apollo link resolver з правилами для формування унікальних ідентифікаторів, отримаємо результат, який наведено у лістингу 13, для прикладу з підрозділу 2.1.

Лістинг 13. Результат роботи Apollo link resolver

```
{
  "sportEvent": {
    "id": "sp1",
    "__typename": "SportEvent",
    "markets": [
      {
        "id": "m1",
        "title": "winner",
        "__typename": "Market",
        "odds": [
          {
            "id": "1",
            "value": 1.5,
            "__typename": "Odd"
          },
          {
            "id": "2",
            "value": 1.75,
            "__typename": "Odd"
          }
        ]
      },
      {
        "id": "m2",
        "title": "First blood",
        "__typename": "Market",
        "odds": [
          {
            "id": "1",
            "value": 1.1,
            "__typename": "Odd"
          },
          {
            "id": "2",
            "value": 2.05,
            "__typename": "Odd"
          }
        ]
      }
    ]  }  }
```

```
{
  "sportEvent": {
    "id": "sp1",
    "__typename": "SportEvent",
    "markets": [
      {
        "id": "sp1~m1",
        "title": "winner",
        "__typename": "Market",
        "odds": [
          {
            "id": "sp1~m1~1",
            "value": 1.5,
            "__typename": "Odd"
          },
          {
            "id": "sp1~m1~2",
            "value": 1.75,
            "__typename": "Odd"
          }
        ]
      },
      {
        "id": "sp1~m2",
        "title": "First blood",
        "__typename": "Market",
        "odds": [
          {
            "id": "sp1~m2~1",
            "value": 1.1,
            "__typename": "Odd"
          },
          {
            "id": "sp1~m2~1",
            "value": 2.05,
            "__typename": "Odd"
          }
        ]
      }
    ]  }  }
```

Передавши отриманий результат до механізму кешування, було отримано результат, який наведений у лістингу 14.

Лістинг 14. Результат модифікованого способу кешування

```
{
  "SportEvent:sp1": {
    "id": "sp1",
    "__typename": "SportEvent",
    "markets": [Market:sp1~m1, Market:sp1~m2] // посилання на об’єкти
  },
  "Market:sp1~m1": {
    "id": "m1",
    "title": "winner",
    "__typename": "Market",
    "odds": [Odd:sp1~m1~1, Odd:sp1~m1~2] // посилання на об’єкти
  },
  "Market:sp1~m2": {
    "id": "m2",
    "__typename": "Market",
    "title": "First blood",
    "odds": [Odd:sp1~m2~1, Odd:sp1~m2~2] // посилання на об’єкти
  },
  "Odd:sp1~m1~1": {
    "id": "1",
    "value": 1.5,
    "__typename": "Odd"
  },
  "Odd:sp1~m1~2": {
    "id": "1",
    "value": 1.75,
    "__typename": "Odd"
  },
  "Odd:sp1~m2~1": {
    "id": "1",
    "value": 1.1,
    "__typename": "Odd"
  },
  "Odd:sp1~m2~2": {
    "id": "1",
    "value": 2.05,
    "__typename": "Odd"
  },
}
```

Було отримано коректно збережений кеш, з контекстними об’єктами. Такий підхід дозволяє правильно оброблювати патч інформації, оскільки ідентифікатори базуються на основі власних ідентифікаторів, а не на порядковому номері в масиві.

2.3. Аналіз алгоритму обробки даних в Apollo link resolver

Основна ідея алгоритму полягає у використані відповідного модифікуючого правила для кожного вкладеного об’єкта. При створенні Apollo link resolver отримає реєстр користувацьких правил, які відповідають за модифікацію об’єктів певного типу.

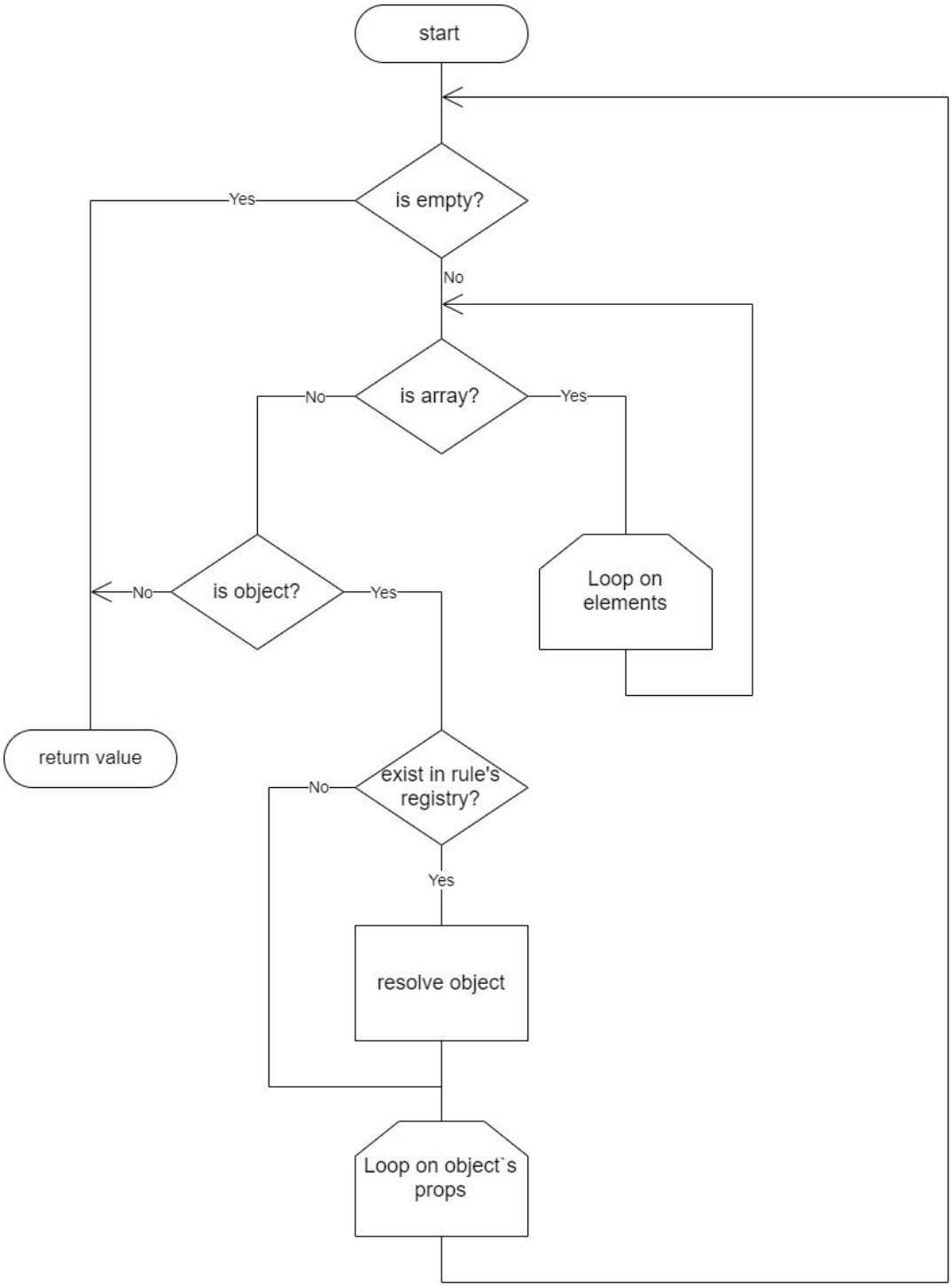


Рис. 10. Блок-схема алгоритму обробки даних в Apollo link resolver

З точки зору реалізації, обробка та модифікація даних є послідовним алгоритмом, який буде виконуватися в процесі роботи Apollo link resolver. Оскільки Apollo link resolver використовується безпосередньо перед terminate link в загальному ланцюгу, то у системі будуть фігурувати об’єкти, модифіковані за даним алгоритмом.

Вхідними даними для алгоритму може бути будь-яка JSON структура. Але в рамках нашого завдання, на вхід даному алгоритму буде подаватися результат виконання запиту з сервера.

Робота алгоритму полягає у рекурсивному переборі вкладених об’єктів та виконанні відповідних для них модифікуючих правил.

2.4. Аналіз роботи правил для модифікації об’єктів

Структури даних, в яких буде використовуватися наш метод, різні, тому зав’язуватися на конкретні ключі не можна. Тому було виділено абстракцію над процесом модифікації одного об’єкту. Такий підхід досить гнучкий та дозволяє користувачеві налаштувати процес модифікації під власні структури даних.

Обробка об’єкті за певними правилами має низку переваг:

– гнучкість;

– простота;

– розділення функціональності.

Правила створюються під конкретні типи об’єктів. Ці типи визначаються полем __typename. Apollo Client автоматично створює поле з такою назвою та в якості значення використовує назву інтерфейсу, яка вказана в GraphQL схемі.

Кожне правило містить в собі наступну інформацію:

– тип об’єкту для якого можна використовувати дане правило;

– функція обробки помилки;

– функція модифікації.

Функція модифікації отримує об’єкт, який необхідно модифікувати та посилання на контекст, в рамках якого необхідно виконати модифікацію. Наприклад, при формуванні поточного ідентифікатора, можна «склеїти» ідентифікатор об’єкту з ідентифікатором контексту. За допомогою таких правил було модифіковано сутності Market і Odd в прикладі з підрозділу 2.1.

При створенні Apollo link resolver необхідно передати набір правил, які організовані в реєстр. Реєстр використовується для створення набору правил, які є необхідними при роботі Apollo link resolver.

Реєстр містить наступну інформацію:

– функцію отримання модифікуючої функції за типом об’єкту (__typename);
– функція додання правила;
– функція видалення правила для конкретного типу об’єктів;
– функцію очистки реєстру.

Використання такого підходу дозволяє розділити виконання обробки даних, які надходять з серверу, за алгоритмом з підрозділу 2.3 від модифікації конкретного об’єкта.

2.5. Висновки до розділу 2

Беручи до уваги усі переваги існуючого механізму кешування даних в бібліотеці Apollo Client було прийнято рішення не змінювати зовнішній інтерфейс роботи з ним. Запропонований спосіб дозволяє модифікувати процес кешування шляхом зміни мережевого шару.

Основним завданням модифікованого способу кешування є надання можливості Apollo Client коректної роботи з контекстно-залежними об’єктами.

Запропонований Apollo Link resolver вбудовується в існуючий ланцюг мережевого шару перед Apollo Link, який взаємодіє з мережею (terminate link), та оброблює результати запитів за вхідними правилами.

Процес виконання обробки результату запиту базується на рекурсивному переборі вкладених об’єктів та застосуванні для їх обробки відповідних правил.

Процес модифікації об’єктів керується заданими розробником правилами. Таким підхід дозволяє не зав’язуватися на конкретних полях та структурах, та надає можливість використовувати даний спосіб для будь-яких схем даних.

Правило модифікую об’єкт на базі отриманого контексту. Таким чином можливе формування глобально-унікального ідентифікатора, що вимагає Apollo Client для коректної роботи вбудованого механізму кешування.

Стосовно програмного забезпечення можна зазначити, що використання розробленої бібліотеки Apollo link resolver надає бібліотеці Apollo Client можливість коректної роботи з контекстно-залежними об’єктами, не вносячи зміни до існуючої кодової бази.

3. ОСОБЛИВОСТІ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ АЛГОРИТМІВ ДЛЯ МОДИФІКОВАНОГО СПОСОБУ КЕШУВАННЯ

3.1. Засоби та технології реалізації програмного забезпечення

На сьогоднішній день, мова JavaScript є стандартом для розробки клієнтських частини веб додатків. Інтерпретація браузерами та відсутність статичної типізації значно пришвидшує розробку додатків на цій мові. Але, з іншого боку, відсутність статичної типізації робить значно складнішим процес налагодження тексту програми.

Typescript – мова, яка підтримує статичну типізацію та дає більші можливості для розробки в об'єктно орієнтованому стиля [13]. Typescript компілюється в звичайний JavaScript, що робить можливим його використання у веб розробці. Статична типізація дозволяє позбавитися більшості помилок ще в процесі розробки, що покращує якість тексту програми, та значно спрощує процес тестування додатку.

Беручи до уваги той факт, що бібліотека Apollo Client та її компоненти розроблені на мові Typescript, було прийнято рішення у використанні цієї мови для написання запропонованої бібліотеки.

3.1.1. *Засоби для збірки бібліотеки*

В якості збірника було використано бібліотеку rollup. Найпопулярнішим збірником JavaScript додатків є webpack. Він має досить широкі можливості для збірки проектів, але для розроблюваної бібліотеки більшість функціональності є надлишковою. Також, в результаті роботи webpack отримаємо файл більших розмірів, що для бібліотеки є критичною характеристикою.

Rollup був обраний за наступними критеріями [14]:

– розмір вихідного файлу;

– швидкість;

– простота налаштування;

– підтримка модулів: CJS, UMD, ESM;

– наявність плагіну для роботи збірки typescript;

– наявність плагінів для мініфікації, стиснення та аналізу отриманого файлу.

Для налаштування rollup було використано наступні плагіни:

– rollup-plugin-sourcemaps [15]. Використовується для для збирання вихідних карт з sourceMappingURL. Це допомагає у налагоджені тексту програми.

– rollup-plugin-node-resolve [16]. Використовується для завантаження сторонніх модулів.

– rollup-plugin-typescript2 [17]. Використовується для транспіляції typescript в JavaScript.

– rollup-plugin-visualizer [18]. Використовується для аналізу зібраної бібліотеки.

– rollup-plugin-terser [19]. Використовується для мініфікації отриманого тексту програми.

– rollup-plugin-gzip [20]. Використовується для стиснення тексту програми, тим самим зменшуючи розмір бібліотеки.

Для налаштування збірника було створено конфігураційний файл.

Лістинг 15. Основна частина з rollup.config.js

```
export default [{
        input: 'src/index.ts',
        output: {
            file: './lib/index.umd.js',
            format: 'umd',
            sourcemap: true,
            name: 'apolloLink.resolver',
            exports: 'named'
        },
        plugins: [
            node(),
            typescriptPlugin({
                typescript,
                objectHashIgnoreUnknownHack: true,
                tsconfig: './tsconfig.json',
                tsconfigOverride: {
                    compilerOptions: {
                        module: "es2015",
```

Продовження лістингу 15

```
                },                },

            }),
            sourcemaps(),
            terser(),
            gzipPlugin()
        ],
    },
    {
        input: 'src/index.ts',
        output: {
            file: './lib/index.esm.js',
            format: 'esm',
            sourcemap: true,
        },
        plugins: [
            node(),
            typescriptPlugin({
                typescript,
                objectHashIgnoreUnknownHack: true,
                tsconfig: './tsconfig.json',
                tsconfigOverride: {
                    compilerOptions: {
                        module: "es2015",
                    },
                },
            }),
            terser(),
            gzipPlugin(),
            sourcemaps(),
            visualizerPlugin({
                sourcemap: true,
                template: "circlepacking"
            }),
        ],
    },
    {
        input: 'lib/index.esm.js',
        output: {
            file: './lib/index.cjs.js',
            format: 'cjs',
            sourcemap: true,
        },
    }
];
```

Для компілятора typescript були задані налаштування, які наведені у лістингу 16.

Лістинг 16. tsconfig.json

```
{
  "compileOnSave": false,
  "buildOnSave": false,
  "compilerOptions": {
    "target": "es5",
    "module": "es6",
    "moduleResolution": "node",
    "importHelpers": true,
    "strict": true,
    "removeComments": true,
    "declaration": true,
    "declarationMap": true,
    "noErrorTruncation": true,
    "noUnusedLocals": true,
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true,
    "baseUrl": "./src",
    "skipLibCheck": true,
    "declarationDir": "./lib",
    "lib": [
      "es6"
    ]
  },
  "exclude": [
    "node_modules",
    "lib"
  ]
}
```

Компілятор typescript з такою конфігурацією буде виконувати транспіляцію typescript файлів, які знаходяться в директорії src. В процесі компіляції буде створено декларативні файли d.ts, які містять інформацію про інтерфейси у відповідних файлах.

3.2. Аналіз архітектури розроблюваної бібліотеки

Розроблена бібліотека складається з 4 основних частин та надає розробнику Apollo Link, та інструменти для створення особистих правил обробки даних:

– ApolloLinkResolver. Основний клас, який створює Apollo Link.

– Алгоритм обробки даних.

– RulesRegistry. Клас для створення реєстру правил, екземпляр якого вимагає ApolloLinkResolver.

– Rule. Клас для створення правил обробки даних.

– ApolloLinkResolver унаслідується від Apollo Link з бібліотеки.

Apollo Link Resolver було розроблено для можливості використання його, як частини мережевого шару для Apollo Client.

RulesRegistry – асоціативний об’єкт, який містить інформацію про типи об’єктів та відповідний йому обробник. Також реєстр надає інтерфейс для наповнення, видалення та очистки правил.

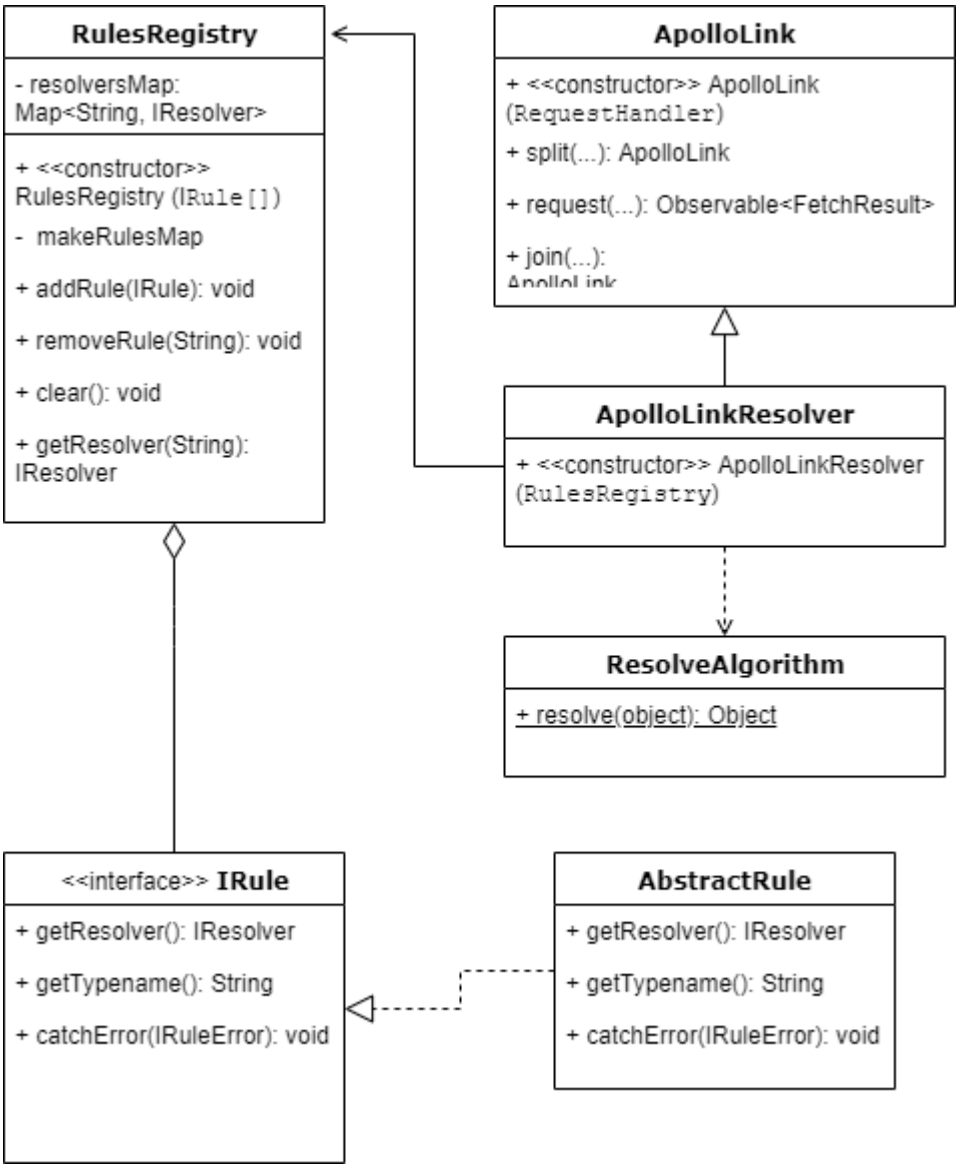


Рис. 11. Діаграма класів

Rule – загальний інтерфейс, який описує правила обробки конкретних типів об’єктів.

Алгоритм обробки включає в себе усі етапи модифікації вхідного об’єкту. Основна ідея полягає у виклику обробників для усіх типів об’єктів в разі наявності їх в реєстрі правил.

3.3. Особливості реалізації розробленого Apollo Link Resolver

Apollo Link Resolver розроблений у стилі afterware. У загальному ланцюгу мережевого шару, Apollo Link Resolver повинен бути вбудований перед termination link. Termination link – це останній обробник в загальному ланцюзі. На практиці, в якості termination link використовується apollo-link-http [21] для передачі GraphQL запиту шляхом http з’єднання, або apollo-link-ws [22] для передачі GraphQL запиту шляхом WebSocket з’єднання.

В загальному, Apollo Link побудований на основі поведінкового патерну проектування Observer, який імплементує механізм підписок. Це дозволяє Apollo Link реагувати на події, які відбуваються в іншому Apollo Link.

Робота Apollo link Resolver полягає в обробці усіх даних, які надходять з наступного Apollo Link. Отримана операція передається наступному Apollo Link та створюється підписка на його результат. Було встановлено власний обробник, а саме виконання основного алгоритму обробки запитів, який буде виконуватися для кожного набору даних, які надходять з наступного (termination) link. Модифікований результат передається попередньому Apollo Link, шляхом відправки даних у спостерігача (observer).

Лістинг 17. Apollo-link-resolver

```
export function makeApolloLinkResolver(
    rulesRegistry: IRuleRegistry
): ApolloLink {
    return new ApolloLink((operation, forward) => {
        return new Observable(observer => {
            if (!forward) {
                observer.next(operation);
                observer.complete();

                return;
            }
```

Продовження лістингу 17

```
            let subscription: Subscription;

            try {
                subscription = forward(operation).subscribe({
                    next: result => {
                        result.data = resolve(
                            rulesRegistry,
                            operation,
                            result.data
                        );

                        observer.next(result);
                    },
                    complete: observer.complete.bind(observer),
                });
            } catch (e) {
                observer.error(e);
            }

            return () => {
                if (subscription) {
                    subscription.unsubscribe();
                }
            };
        });
    });
}
```

3.4. Особливості реалізації алгоритму обробки даних

На вхід до apollo-link-resolver потрапляють дані з мережі. Отже, алгоритм повинен вміти коректно оброблювати усі типи даних. Блок-схема роботи даного алгоритму була представлена на рис. 12.

Лістинг 18. Алгоритм обробки даних

```
function resolveObject(
    rulesRegistry: IRuleRegistry,
    operation: Operation,
    value: any,
    parent?: object
): any {
    const resolver =
rulesRegistry.getResolver(value['__typename']);
    let resolvedObject = value;
    if (resolver) {
        resolvedObject = resolver(operation, value, parent);
    }
    for(let prop in resolvedObject) {
        if(resolvedObject.hasOwnProperty(prop)) {
            resolvedObject[prop] = resolve(rulesRegistry,
operation, resolvedObject[prop], resolvedObject);
        }
    }
    return resolvedObject;
}
```

Продовження лістингу 18

```
function resolveArray(
    rulesRegistry: IRuleRegistry,
    operation: Operation,
    values: any[],
    parent?: object
): any {
    return values.map(value => resolve(rulesRegistry, operation,
value, parent));
}
export function resolve(
    rulesRegistry: IRuleRegistry,
    operation: Operation,
    value: any,
    parent?: object
): any {
    if (!value) return value;

    if (Array.isArray(value))
        return resolveArray(rulesRegistry, operation, value,
parent);

    if (typeof value === 'object')
        return resolveObject(rulesRegistry, operation, value,
parent);

    return value;
}
```

Функція resolve перевіряє наявність вхідного значення та його тип. В разі відсутності вхідного значення відбувається вихід. Якщо тип вхідного значення примітив – відбувається його повернення. Якщо вхідним значенням є масив даних, виконуємо функцію обробки масивів (resolveArray). Якщо вхідним значенням є об’єкт – виконуємо функцію оброби об’єктів (resolveObject).

Функція обробки масивів resolveArray послідовно виконує функцію resolve для кожного елементу масиву.

Функція обробки об’єктів resolveObject отримує тип об’єкту (typename) та намагається отримати правило обробки об’єктів даного типу з реєстру. Якщо привило існує, викликаємо його, передавши операцію, оброблюваний в даний момент об’єкт та посилання на контекст. Наступним кроком відбувається послідовний виклик функції resolve для кожного значення в ключах об’єкту.

Результатом рекурсивної обробки вхідного значення є об’єкт, який був сформований результатами виконання правил для відповідних типів. В разі відсутності правила для об’єкту повертається вхідне значення.

3.5. Особливості створення правил обробки даних

Правило відповідає за обробку об’єктів конкретного типу. За допомогою правил регулюється процес обробки вхідного значення. Встановлені розробником правила будуть використанні у основному алгоритмі обробки даних. Наприклад, apollo-link-resolver з пустим реєстром взагалі не модифікую вхідне значення.

Основна вимога при створенні правил є збереження схеми даних. Тобто значення в полі не можуть змінювати тип. Також в об’єкт не можуть бути додані додаткові поля чи видалені існуючі. В разі порушення цього правила дані не пройдуть валідацію зі схемою на етапі збереження у кеш.

Таке обмеження не заважає створювати правила для модифікації ідентифікаторів, а, отже, не заважає основній ідеї розробки.

Лістинг 19. Інтерфейс Rule

```
interface IRuleError<Input> {
    operation: Operation,
    message?: string;
    value?: Input;
    parent?: any;
}

export type IResolver<Input = any, Output = any> = (operation:
Operation, value: Input, parent?: any) => Output

export interface IRule<Input = any, Output = any> {
    catchError(error: IRuleError<Input>): void;
    getTypename(): string;
    getResolver(): IResolver<Input, Output>
}
```

В розробленій бібліотеці було створено абстрактний клас правила, в якому реалізовано стандартний механізм обробки помилки. Стандартним механізмом обробки помилки є вивід інформації про помилку у консоль

розробника. Користувач бібліотеки може створювати власні правила шляхом наслідування від класу IRule або AbstractRule.

Лістинг 20. AbstractRule

```
export abstract class AbstractRule<Input, Output> implements
IRule<Input, Output> {
    catchError({
                    operation,
                    value,
                    message,
                    parent,
                }: IRuleError<Input>): void {
        console.error(
            `Normalization error:
                        operation: ${JSON.stringify(operation,
null, 2)}
                        message: ${message}
                        typename: ${this.getTypename()}
                    `
        );
    }

    abstract getTypename(): string;
    abstract getResolver(): (
        operation: Operation,
        value: Input,
        parent?: any
    ) => Output;
}
```

Використовуючи правила можна створювати унікальні ідентифікатори для контекстно-залежних сутностей. Для прикладу розглянемо правила для сутностей Market і Odd з лістингу 8.

Лістинг 21. Правила для Market і Odd

```
class MarketRule extends
AbstractRule<Partial<API.SportEventMarket>, any> {
    getTypename(): string {
        return 'SportEventMarket';
    }
    getResolver() {
        return (
            operation: Operation,
            value: Partial<API.SportEventMarket>,
            parent: any
        ) => {
            const sportEventId =
                get(parent, 'id') || get(parent, 'sportEvent.id');
            const marketId = get(value, 'id');

            if (!sportEventId || !marketId) {
                this.catchError({ operation, value, parent });
```

Продовження лістингу 21

```
                return value;
            }

            return {
                ...value,
                id: `${sportEventId}${idPathDivider}${marketId}`,

            };
        };
    }
}

class OddRule extends AbstractRule<Partial<API.Odd>, any> {
    getTypename(): string {
        return 'Odd';
    }
    getResolver() {
        return (operation: Operation, value: Partial<API.Odd>,
parent: any) => {
            const marketId = get(parent, 'id');
            const oddId = get(value, 'id');

            if (!marketId || marketId === idPathDivider || !oddId)
{
                this.catchError({ operation, value, parent });

                return value;
            }

            return {
                ...value,
                id: `${marketId}${idPathDivider}${oddId}`,
            };
        };
    }
}
```

Використовуючи дані правила, результатом роботи apollo-link-resolver будуть дані з лістингу 13. Тобто отримуємо коректну роботу кешу для контекстних сутностей Market та Odd.

3.6. Висновки до розділу 3

Для реалізації програмного забезпечення було створено архітектуру, яка складається з 4-ох основних частин: Apollo link resolver, алгоритм обробки даних, правила обробки об’єктів конкретного типу та механізм наповнення правил. В якості базового інтерфейсу було обрано інтерфейс Apollo Link, що дало змогу застосовувати розроблену бібліотеку в якості ланки в загальному ланцюзі мережевого шару Apollo Client.

В якості мови програмування було обрано typescript. Даним вибір був зумовлений наступними факторами:

– статична типізація;
– бібліотека Apollo Client та всі її частини розроблені на мові typescript;
– процес налагодження тексту програми;
– зменшення кількості тестів;
– розроблена бібліотека використовується у JavaScript середовищі.

В якості збірника бібліотеки було використано rollup. Налаштування даного збірника наведено у лістингу 15. Збірник був налаштований для формування бібліотеки для 3-ох середовищ використання:

– index.esm.js – для використані бібліотеки на стороні клієнта у JavaScript додатках;
– index.cjs.js – для використання на стороні сервера при Server side rendering для Node.js
– index.umd.js – універсальних формат для використані у будь-якому середовищі JavaScript додатків.

Зібраний текст програми бібліотеки був мініфікований та стиснений.

Розроблений Apollo link resolver виконує модифікацію отриманих даних з мережі на основі заданих розробником правил. Розроблена бібліотека надає розробнику класи та інтерфейси для створення власних правил.

Також слід зазначити, що було проведено інтеграцію розробленої бібліотеки в існуючий проект. В результаті було отримано коректну роботу кешу для контекстно-залежних сутностей.

4. АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

4.1. Тестування розробленого програмного забезпечення

Тестування розробленої бібліотеки відбувалося на кожному з етапів розробки. Оскільки була повністю відома і доступна структура тексту програми, було прийнято рішення використовувати метод білого ящика [23].

В основі методу білого ящика лежать unit тести. Було проаналізовано існуючи методи написання unit тестів в середовищі JavaScript. Вибір бібліотеки jest [24] був замовлений наступними факторами:

– наявність хорошої документації;

– підтримка мови typescript

– підтримка snapshots;

– наявність механізму перевірки покриття тексту програми тестами (coverage);

– детальний опис помилки та місця її виникнення.

Лістинг 22. Приклад unit тесту в системі

```
it('Correct apply rule on test data', done => {
    const mockTerminationLink = new ApolloLink(operation =>
        Observable.of({ data: mockData })
    );

    const link = makeApolloLinkResolver(
        new RulesRegistry([new MarketRule(), new OddRule()])
    ).concat(mockTerminationLink);

    execute(link, { query }).subscribe(result => {
        expect(result).toMatchSnapshot();
        done();
    });
});
```

Jest надає механізм перевірки покриття тексту програми тестами. Було протестовано наступні частини тексту програми [26]:

– строкове покриття;

– покриття рішень;

– покриття умов;

– цикли;

– шляхи;

– покриття потоків даних.

Результат такого тестування наведено на рис. 14.

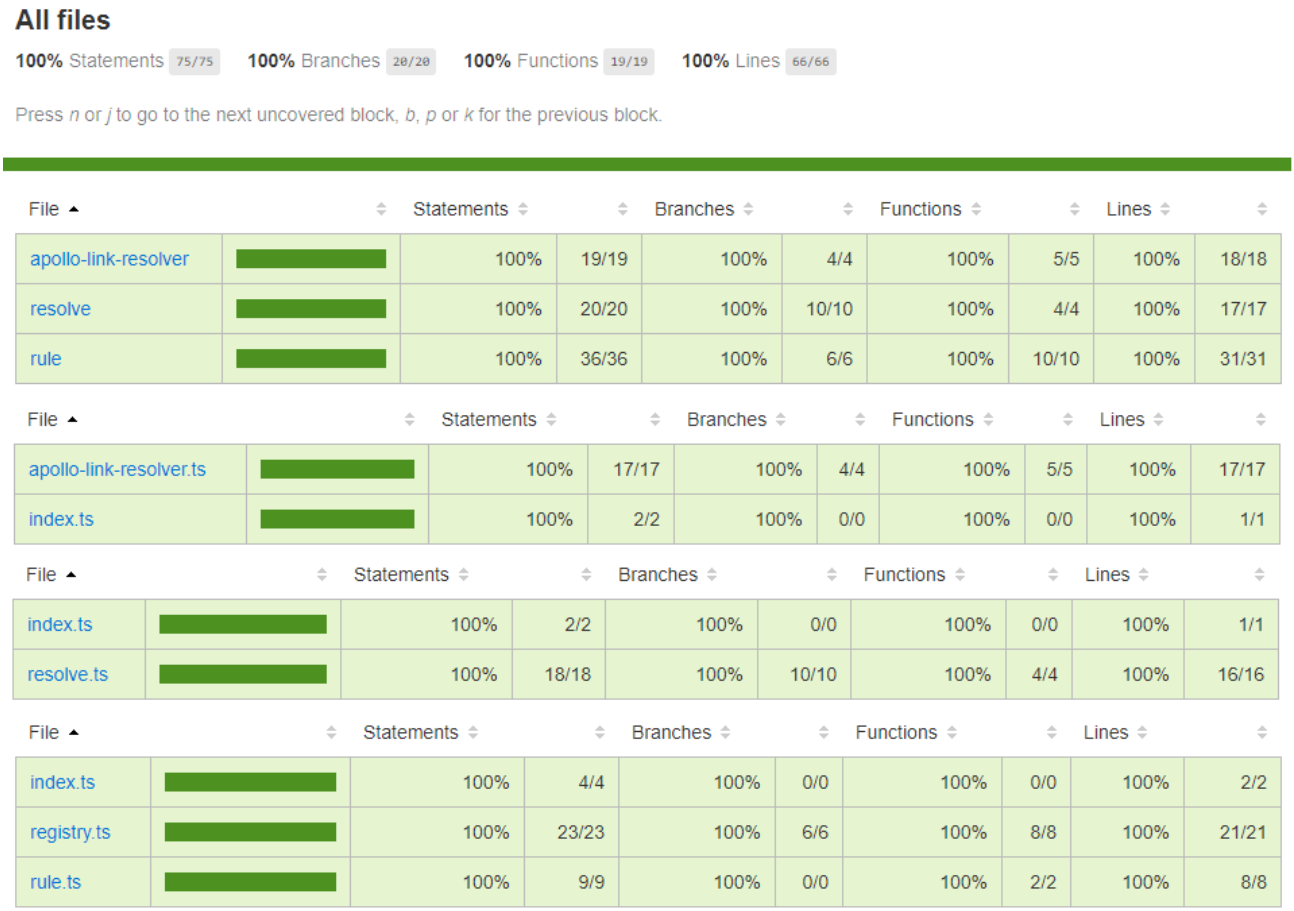


Рис. 12. Покриття тестами

При написані тексту програми було використано бібліотеку TSLint [25]. Інструмент було використано для статичного аналізу, який перевіряє текст програми на наявність функціональних та синтаксичних помилок. Також можна задати певний набір правил для оформлення тексту програми. Інструмент був створений як для запобігання помилок так і для підтримки єдиного стилю при написані тексту програми.

Оскільки розроблювана бібліотека є модифікацією бібліотеки Apollo Client, було прийнято рішення дотримуватися правил написання тексту програми, які зазначені для цієї бібліотеки.

В кінці розробки було проведено тестування методом чорного ящика. Цей тест є досить важливим, оскільки абстрагується від тексту програми та концентрується на перевірці якості роботи програмного забезпечення з точки зору кінцевого користувача. Результат виконання даного тесту показав можливість зручної та швидкої інтеграції розробленої бібліотеки в існуючі проекти.

За результатами проведеного тестування можна сформувати звіт, що розроблена бібліотека не містить критичних помилок, які можуть негативно вплинути на якість виконання програмного забезпечення під час роботи. Отже, дана розробка може використовуватися у production проектах.

4.2. Аналіз розробленого програмного забезпечення

Після завершення розробки програмного забезпечення було проведено детальний аналіз отриманих результатів. Розроблена бібліотека відповідає усім вимогам якості написання тексту програми. Для збереження єдиного стилю написання тексту програми були використані правила для TSLint, які прийняті для Apollo Client та усіх його складових.

Розроблене програмне забезпечення на 100% покрите тестами та не має критичних помилок.

Розроблена бібліотека:

- Надає інтерфейс для створення власних правил обробки об’єктів відповідних типів. Надає абстрактний клас правила, в якому реалізовано обробку помилок за замовченням.
- Надає клас для створення та управління реєстром правил.
- Надає функцію для створення Apollo Link, яка реалізує механізм модифікації отриманих даних на основі заданих розробником правил.

Механізм обробки даних за правилом досить гнучкий, та може використовуватися для багатьох задач. Модифікуючий метод правила отримує посилання на контекст. За допомогою цього механізму було вирішено проблему збереження контекстно-залежних сутностей у кеш.

Основним обмеженням даного підходу є неможливість внесення змін до схеми отриманих даних. Наявність такого обмеження було передбачувано, оскільки при аналізі роботи механізму кешування Apollo Client було досліджено етап валідації даних за схемою. Дане обмеження не заважає вирішенню проблеми збереження контекстно-залежних сутностей у кеш, оскільки запропонований спосіб не передбачає зміну схеми даних.

Використання розробленої бібліотеки можливе як на стороні клієнта так і на стороні сервера. Apollo Client підтримує Server side rendering, отже, розроблена бібліотека також повинна надавати таку можливість. Тому розроблена бібліотека була зібрана у наступних форматах: cjs, esm, umd.

При розробці бібліотеки для веб, критичним моментом є розмір отриманого файлу. Це впливає на швидкість завантаження веб додатку. Розроблена бібліотека була проаналізована за допомогою rollup-plugin-visualizer.

Розмір розробленої бібліотеки у мініфікованому стані складає 1.68 Kb. Також в процесі збірки було створено gzip [27] версію бібліотеки, розмір якої складає 779 байт.

Розроблена бібліотека не має внутрішніх залежностей від інших бібліотек. Але в проекті, який буде використовувати розроблену бібліотеку, повинна бути встановлена залежність apollo-link [28].

Рис. 13. Аналіз bundle розробленої бібліотеки

В ході проектування програмного забезпечення було прийнято рішення не змінювати зовнішнє API роботи зі стандартним механізмом кешування даних бібліотеки Apollo Client. Проблему коректної роботи кешу при наявності контекстно-залежних сутностей було вирішено шляхом модифікації мережевого шару шляхом додання Apollo link resolver в загальний ланцюх Apollo Link. Таке рішення дозволяє використовувати розроблену бібліотеку не змінюючи існуючу кодову базу.

Отже, розроблене програмне забезпечення повністю вирішує проблему роботи Apollo Client з контекстно-залежними сутностями. Також текст програми покритий тестами та відповідає загальному стилю написання тексту програми, який прийнятий для продуктів Apollo.

4.3. Вдосконалення розробленого програмного забезпечення

Розроблене програмне забезпечення повністю вирішує проблему збереження контекстно-залежних сутностей в бібліотеці Apollo Client.

Але в процесі аналізу роботи Apollo cache було виявлено ще одну суттєву проблему. Проблема полягає у інвалідації даних у кеші. Проблема є глобальною, але особливо помітна при роботі зі списками та пагінацією.

Для прикладу, розглянемо звичайний список по 5 елементів на сторінку. Існує 2 GraphQL mutation для видалення та додавання елементу відповідно. Після видалення, або додавання елементу, дані, які були збережені в кеші для сторінки зі списком, є невалідними. Проблема посилюється тим фактом, що користувачеві потрібно повністю обновити сторінку для того, щоб побачити зміни у списку. Це зумовлено тим фактом, що Apollo Client закешував запит. Це означає, що запит на отримання списку не буде відправлений на сервер, а будуть повернені дані з кешу, які є вже не коректними.

Apollo client дозволяє вирішити цю проблему декількома способами, але усі з них мають негативний ефект [29]:

Не використовувати кешування

При виконанні запиту розробник може вказати fetchPolicy network-only. Це означає, що дані запиту не будуть зберігатися у кеші. Це вирішує проблема, але тільки при повторному запиті списку (можна без перезавантаження сторінки). Але ми втрачаємо швидкість роботи додатку, та збільшуємо навантаження на сервер.

Повторне завантаження запитів

Apollo client дозволяє вказати масив запитів, які необхідно виконати повторно після виконання mutation. Такий спосіб, також, вирішує проблему. Але, якщо користувач виконав 10 переходів по сторінкам, було виконано 10 однакових запитів, але з різними аргументами. В такому випадку, якщо вказати цей запит для повторного виконання після mutation, до мережі буде одночасно надіслано 10 запитів.

Зміна даних в кеші

Apollo client, також надає API для зміни даних запиту безпосередньо у кеші. Тобто, після видалення елементу списку, сервер повертає ідентифікатор видаленої сутності. Розробник виконує запит до кешу для отримання даних по запиту, та видаляє потрібний елемент. Проблема полягає в тому, що після видалення на екрані користувач отримає 4 елементи, хоча на решті сторінок по 5 елементів. Також проблема є при додаванні нових елементів. Користувач отримає 6 елементів на сторінки замість 5-ти. Також, при додаванні елементу, потрібно повторити логіку сортування елементів списку, а це не завжди можливо виконати на стороні клієнта не маючи всіх даних.

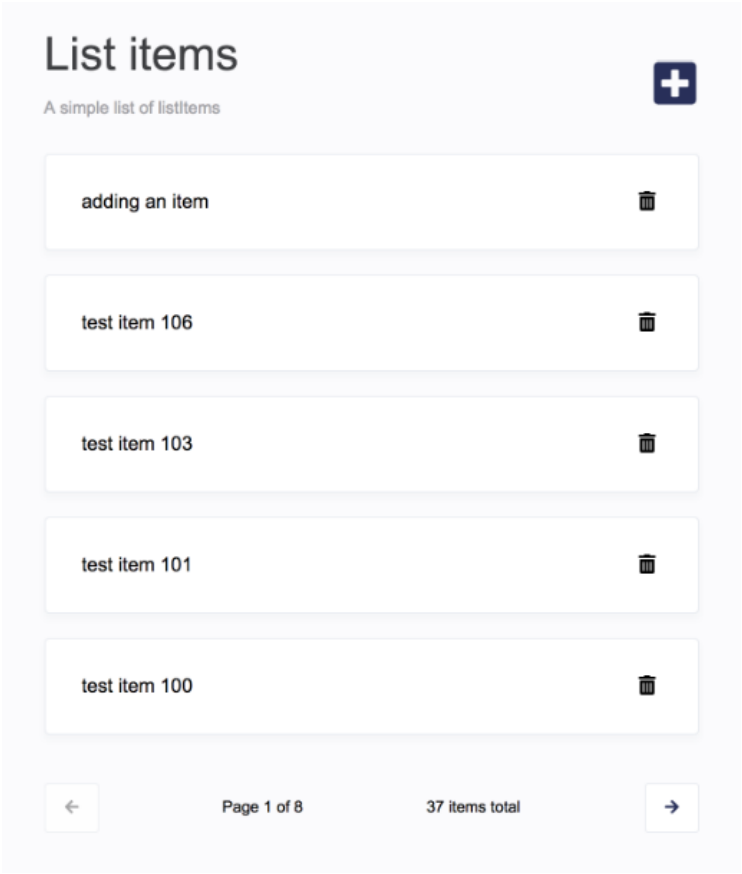


Рис. 14. Список елементів

4.4. Висновки до розділу 4

У даному розділі було проаналізовано методи тестування розробленого програмного забезпечення. Для тестування використовувалися наступні інструменти: TSLint, Jest та Jest coverage. Результати тестування наведені на рис. 14. За результатами можна сформувати звіт, що розроблене програмне забезпечення покрите тестами на 100% та відповідає усім вимогам якості написання тексту програми. Також було прийнято рішення дотримуватися правил написання тексту програми, які зазначені для бібліотеки Apollo та всіх її складових. Тому було використано такі ж налаштування для TSLint.

Розроблена бібліотека може використовуватися як на стороні сервера (nodejs) так і на стороні клієнта. Для цього було зібрано декілька екземплярів бібліотеки для різних середовищ:

– index.esm.js – для клієнтських додатків;

– index.cjs.js – для nodejs середовища;

– index.umd.js – універсальний варіант для будь-якого JavaScript середовища.

Розроблена бібліотека не має жодних залежностей від сторонніх бібліотек. Але в проекті, який буде використовувати розроблену бібліотеку, має бути встановлений пакет Apollo-link.

Було детально проаналізовано розмір отриманої бібліотеки. В результаті, мініфокована версія – 1.68 Kb, а gzip версія – 779 байт. Це дозволяє використовувати розроблену бібліотеку для веб додатків, де розмір залежностей є критичним, оскільки впливає на швидкість завантаження ресурсу.

5. ПОБУДОВА БІЗНЕС-МОДЕЛІ

5.1. Аналіз проблеми

Для роботи з екосистемою GraphQL наразі існує дві основних бібліотеки Apollo та Relay. За останні кілька років на ринку сформувалося чітке домінування бібліотеки Apollo. Навколо цієї бібліотеки було сформовано екосистему, яка включає в себе Apollo Client, Apollo server, Apollo engine, schema validator, та багато інших бібліотек. Також була сформована потужна громада, яка активно підтримує проект.

Проаналізувавши бібліотеку Apollo Client було виявлено проблему, яка, в деяких випадках, робить неможливим її використання. Проблема стосується коректної роботи з контекстно-залежними сутностями. А саме, відбувається перетирання сутностей в кеші. Це відбувається без жодних помилок, що робить процес виявлення таких помилок досить складним.

Також, слід зауважити, що частіше за все, проблема виявляється в середині циклу розробки програмного забезпечення. Це призводить до ситуації, коли необхідно вирішувати проблему будь-якими методами, оскільки переписувати розроблену частину продукту, використавши іншу бібліотеку для роботи з GraphQL, стає надто дорого та займає багато часу.

Було проаналізовано механізм роботи Apollo cache та сформовано звіт про неможливість коректної роботи з контекстно-залежними об’єктами. Також, на даний момент не існує жодного стороннього рішення.

Слід зазначити, що зміна публічного API роботи з бібліотекою Apollo client буде потребувати зміни існуючої кодової бази, це, в свою чергу, буде перешкодою для використання сторонньої бібліотеки для вирішення проблеми кешування контекстно-залежних сутностей.

5.2. Зацікавлені сторони

В процесі створення бізнес-моделі проекту було сформовано групи зацікавлених в реалізації проекту сторін:

– менеджери проектів;

– розробники (громада);

– компанія Apollo.

Було проведено детальний аналіз кожної зацікавленої групи та сформовано звіт з наступною інформацією:

– рівень зацікавленості в проекті по 10-ти бальній шкалі;

– рівень впливу на проект по 10-ти бальній шкалі;

– методи роботи із зацікавленою групою.

Результат такого аналізу представлено в табл. 1.

Таблиця 1

Аналіз зацікавлених сторін

Зацікавлена сторона	Рівень зацікавленості	Рівень впливу	Методи роботи
Менеджери проектів	7/10	1/10	Донесення інформації щодо існуючої проблеми та розробленої бібліотеки для її вирішення на форумах та популярних ресурсах.
громада	8/10	4/10	Популяризація розробленої бібліотеки такими каналами як: Stask overflow, medium, github.
Компанія Apollo	6/10	8/10	Прямі пропозиції керівництву компанії.

5.3. Аналіз рішення проблеми

В якості рішення вищевказаної проблеми було прийнято рішення щодо створення окремої бібліотеки, яка буде містити в собі процес обробки даних за вказаними клієнтом правилами.

Таким чином бібліотека надає набір інтерфейсів та класів для формування правил та створення обробника (Apollo Link). Розробник має помістити отриманий обробник в загальний ланцюг обробки запитів Apollo Link. В результати таких дій, Apollo Client отримає змогу коректно працювати з контекстно-залежними сутностями.

Розроблена бібліотека на змінює існуючі публічні API бібліотеки Apollo Client. Отже, не потребує модифікації існуючої кодової бази проекту.

5.4. Бізнес-продукт. Основні характеристики бізнес-продукту

Результатом виконання магістерської дисертації є програмне забезпечення, яке вирішує проблему збереження контекстно-залежних сутностей в найпопулярнішому клієнту для GraphQL – Apollo client.

За результатом детального аналізу існуючого механізму кешування було сформовано звіт про неможливість коректної роботи з контекстно-залежними об’єктами. Також, пошук існуючих рішень проблеми не привів до позитивних результатів.

Отже, було прийнято рішення щодо розробки власного бізнес-продукту для вирішення вищезгаданої проблеми. Таким бізнес продуктом є розроблена бібліотека Apollo Link Resolver.

В основі розробленої бібліотеки є процес модифікації мережевого шару бібліотеки Apollo Client. Бібліотека надає інтерфейс та класи для формування власних правил обробки певних типів сутностей та Apollo Link, який має бути вбудований в існуючий мережевий шар.

Серед основних переваг такого рішення слід визначити:

– відсутність сторонніх залежностей;
– розміри розробленої бібліотеки у gzip варінті складає 779 байт;
– не змінює існуюче API бібліотеки Apollo Client та не потребує заміни існуючої кодової бази;
– повністю вирішує проблему роботи з контекстно-залежними сутностями.

Підсумувавши вищезазначене можна сформувати звіт, що кінцевий користувач бібліотеки отримає повністю готове та відтестоване рішення для бібліотеки Apollo client, яке вирішує проблему роботи з контекно-залежними сутностями. Це дозволяє суттєво зменшити час на пошук або розробку власного рішення, яке може бути використане у production.

5.5. Конкурентні переваги продукту

На разі не існує сторонніх бібліотек для вирішення виявленої проблеми. Отже, розроблена бібліотека пропонує унікальне рішення на ринку.

Розроблена бібліотека не змінює існуюче API бібліотеки Apollo Client, не залежить від сторонніх бібліотек та займає лише 779 байт. Також Apollo Link Resolver має підтримку усіх JavaScript середовищ.

5.6. Клієнти. Сегменти ринку споживання

Для досягнення максимальної ефективності діяльності команди проекту слід чітко сформувати портрет клієнта та провести сегментацію ринку клієнтів. Процес сегментації дозволяє виділити однорідні грипу користувачів, зацікавлених в запропонованому продукті.

В процесі аналізу було виявлено два клієнти і, відповідно до цього, буде сформовано дві окремі пропозиції.

Першим можливим клієнтом є компанія Apollo. Аналізуючи роботу компанії було виявлено, що основним їх направленням є продукти для екосистеми GraphQL. Компанія має велику кількість власних розробок в цьому напрямі, а також, володіє великою кількістю поглинутих бібліотек. Компанія знаходить цікаві бібліотеки та рішення для екосистеми GraphQL та формує пропозиції щодо їх придбання. Отже, компанія Apollo є потенційним клієнтом запропонованого продукту.

Ще одним клієнтом є розробники програмного забезпечення. Було проведено сегментація ринку розробників за двома категоріями:

– тип проекту (комерційний /не комерційний);
– можливість купити ліцензію на використання запропонованого продукту.

Згідно з вказаними критеріями, можна сформувати 2 групи потенційних клієнтів:

– користувачі, які приймають участь у розробці комерційного програмного забезпечення;
– користувачі, які використовують запропоновану бібліотеку для створення власних некомерційних програмних продуктів.

Перша категорія формує основне джерело прибутку компанії. Друга група дозволяє популяризувати рішення та сформувати громаду.

5.7. Унікальна цінність пропозиції

Запропонований продукт надає готове рішення проблеми роботи з контекстно-залежними сутностями у бібліотеці Apollo Client. На сьогоднішній день це унікальна пропозиція на ринку, оскільки не має жодних готових рішень, які здатні вирішити цю проблему.

Запропонована бібліотека не змінює існуюче API Apollo Client, що дозволяє швидко почати використання, не замінуючи існуючу кодову базу.

5.8. Доходи та витрати

При аналізі клієнтів у підрозділі 5.6 було прийнято рішення у формуванні двох варіантів отримання прибутку:

– продаж розробки для компанії Apollo;
– отримання прибутку шляхом продажу ліцензії на використання.

Для кожного з варіанту було розроблено відповідні моделі підрахунку доходів та витрати. Підрахунок доходів та витрат грошових коштів було проведено у доларах США (USD). Також витрати та доходи були

підраховані за умови того, що проект буде реалізовуватися на території України.

Продаж розробки компанії Apollo

Такий варіант має за мету створення продукту та повний його продаж. Для розробки бібліотеки було використано правила написання тексту програми, які прийняті у компанії Apollo. Це дозволило притримуватися однакового стилю написання тексту програми.

Витрати було підраховано з урахуванням того, що на реалізації проекту знадобиться 1 місяць та 2 виконавця, серед яких один розробник та один менеджер з продажу. При такому варіанті будемо мати наступні витрати:

– на юридичні послуги;

– на комп’ютерну техніку та програмне забезпечення;

– на засоби зв’язку;

– на заробітну плату працівникам;

– податок.

Загальні витрати на проект представлені в табл. 2.

Таблиця 2

Загальні витрати на проект при продажі компанії Apollo

Вид витрат	Вартість ($)
Юридичні послуги	350
Комп’ютерна техніка та ПЗ	2000
Засоби зв’язку	50
Заробітна плата	3000
Податок	300

Враховуючи вищезазначені витрати, кінцева вартість проекту складає 5700$.

Ціна запропонованого продукту для Apollo Client складає 20 000$.

Отже, прибуток запропоновано проекту складає 14 300$.

Продаж ліцензій на використання бібліотеки

Витрати на проект було розділено за двома категоріями: стартові та щомісячні.

До стартових витрат відносяться:

– юридичні послуги;

– комп'ютерна техніка та програмне забезпечення;

– ріелторські послуги, меблі та канцелярія;

– транспортні витрати.

Щомісячні витрати на проект було розділено на наступні групи:

– зарплата за посадами;

– оренда офісного приміщення;

– абонентська плата за засоби зв'язку;

– податки;

– непередбачені витрати.

Загальна модель витрат представлена в табл. 3.

Таблиця 3

Загальні витрати на проект при продажі ліцензій

Вид витрат	Вартість ($)
Стартові	
Юридичні послуги	350
Комп'ютерна техніка та ПЗ	3000
Ріелторські послуги, меблі та канцелярія	2100
Транспортні витрати	150

Продовження табл. 3

Щомісячні	
Зарплата за посадами	3500
Оренда офісного приміщення	600
Абонентська плата за засоби зв’язку	50
Податки	300
Непередбачені витрати	300

Таким чином, необхідна сума для старту складає 5600$, а щомісячні витрати складають 4750$.

Вартість ліцензії на використання запропонованої бібліотеки складає 2$. Отже, для того, щоб проект був прибутковим необхідно продати мінімум 2375 ліцензій щомісяця.

5.9. Висновки до розділу 5

Пропонується унікальний продукт на ринку, який здатен повністю вирішити існуючу проблеми в найпопулярнішій бібліотеці для GraphQL – Apollo Client. Проблема полягає у збережені контекстно-залежних сутностей у кеш. Це призводить до перетирання даних.

Було проаналізовано клієнтів та сформовано дві бізнес моделі. Перша – базується на продажі розробленої бібліотеки для компанії Apollo. Друга - на продажі ліцензій на використання бібліотеки кінцевими користувачами. Витрати та прибуток обох варіантів було підраховано у підрозділі 5.8.

Цінність пропозиції. Продукт, що пропонується, набір інструментів для вирішення проблеми. Продукт призначений для вирішення критичної помилки у бібліотеці Apollo Client. Запропонована бібліотека повністю вирішує проблему роботи з контекстно-залежними об’єктами не змінюючи

існуюче API. Запропонований продукт повністю протестований та підходить для будь-якого JavaScript середовища.

ВИСНОВКИ

Було проаналізовано роботу найпопулярнішої бібліотеки для роботи з GraphQL – Apollo Client. В ході аналізу було виявлено критичну помилку, яка, в деяких випадках, робить неможливим її подальше використання. Проблема полягає у процесі кешування контекстно-залежних сутностей, яка призводить до перетирання даних.

Було прийнято рішення щодо розроблення бібліотеки, яка повністю вирішує знайдену проблему. Основною вимогою до розробленого продукту було збереження існуючого API бібліотеки Apollo Client. Це дозволить використовувати бібліотеку не змінюючи існуючу кодову базу проекту.

Для реалізації програмного забезпечення було прийнято рішення вирішити проблему кешування даних шляхом модифікації мережевого шару. Основна ідея полягає у створені глобально-унікальних ідентифікаторів для кожної сутності.

Було розроблено архітектуру бібліотеки, яка дозволяю розробнику створювати власні правила модифікації для конкретних типів об’єктів. Такий підхід дозволяє позбавитися залежності від конкретних полів в сутності та надає можливість використовувати запропоновану бібліотеку для різних схем даних.

Рішенням є створення Apollo Link, який буде вбудовуватися в мережевий шар проекту перед відправленням запиту до мережі. Таким чином, запропонований Apollo Link Resolver буде отримувати дані з мережі та рекурсивно модифікувати їх у відповідності з правилами розробника. В момент виклику модифікуючого методу правила буде передано контекст виконання. Саме за допомогою контексту відбувається створення унікальних ідентифікаторів.

Розроблена бібліотека не має залежносте від сторонніх бібліотек, та може використовуватися у будь-якому JavaScript середовищі. Загальний розмір бібліотеки для веб додатків складає 779 байт.

За результатами тестування та аналізу кінцевого програмного забезпечення можна сформувати звіт, що розроблений продукт не має критичних помилок, які можуть привести до виключних ситуацій, та може використовуватися у production. Отриманий текст програми відповідає усім правилам написання та оформлення, які зазначені компанією Apollo.

За результатами проведеної роботи було сформовано та побудовано дві бізнес моделі кінцевого продукту. Перша основа на повному продажі розробки для компанії Apollo. А друга базується на розповсюджені ліцензій на використання бібліотеки. Розроблений програмний продукт є комерційно привабливий та має унікальну ціннісну пропозицію.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Banks, A. Learning GraphQL: Declarative Data Fetching for Modern Web Apps [Text] / Alex Banks. — Sebastopol : O'Reilly Media, 2018. — 198 p.
2. Wieruch, R. The Road to GraphQL [Text] / Robin Wieruch. — Independently published, 2018. — 352 p.
3. Arnaud, L. The Design of Web APIs [Text] / Lauret Arnaud. — 1st edition. — New York : Manning Publications, 2019. — 392 p.
4. Lendon, Z. What is GraphQL? [Електронний ресурс] / Zach Lendon. — Режим доступу: https://opensource.com/article/19/6/what-is-graphql. — Дата доступу: 12.07.2019.
5. Niedringhaus, P. What is GraphQL, really? [Електронний ресурс] / Paige Niedringhaus. — Режим доступу: https://medium.com/@paigen11/what-is-graphql-really-76c48e720202. — Дата доступу: 15.07.2019.
6. Gurumoorthy, P. GraphQL [Електронний ресурс] / Paige Gurumoorthy. — Режим доступу: https://devopedia.org/graphql. — Дата доступу: 15.07.2019.
7. Tiwari, A. How to GraphQL [Електронний ресурс] / Ankit Tiwari. — Режим доступу: https://www.howtographql.com/basics/0-introduction/. — Дата доступу: 20.07.2019.
8. Pandya, D. GraphQL Concepts Visualized [Електронний ресурс] / Dhaviat Pandya. — Режим доступу: https://blog.apollographql.com/the-concepts-of-graphql-bc68bd819be3. — Дата доступу: 02.09.2019.
9. Dedmon, M. Getting started with Apollo Client and GraphQL [Електронний ресурс] / Morgan Dedmon. — Режим доступу: https://www.slideshare.net/MorganDedmon/getting-started-with-apollo-client-and-graphql. — Дата доступу: 08.09.2019.

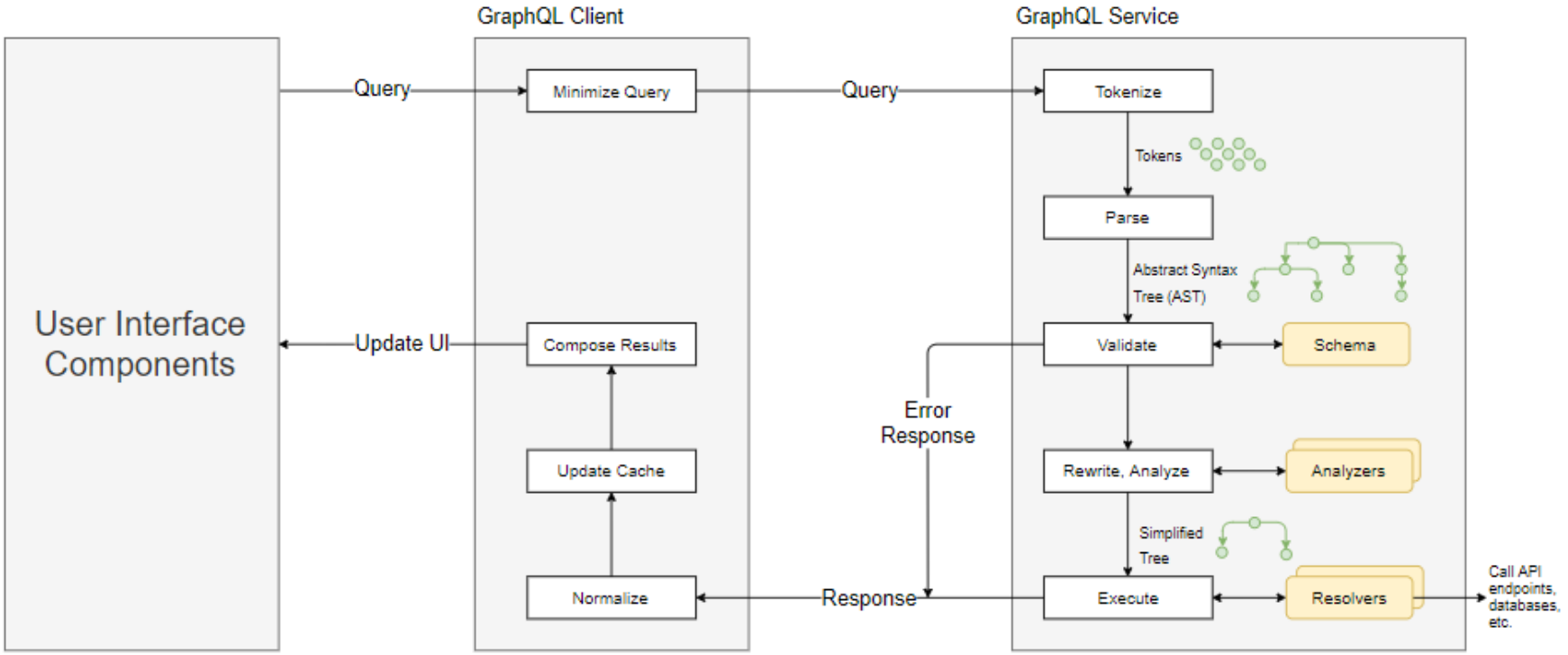
10. Apollo, Network layer [Електронний ресурс] / Apollo. — Режим доступу: https://www.apollographql.com/docs/react/networking/network-layer/. — Дата доступу: 20.09.2019.
11. Apollo, Configuring the cache [Електронний ресурс] / Apollo. — Режим доступу: https://www.apollographql.com/docs/react/caching/cache-configuration/. — Дата доступу: 22.09.2019.
12. Newman, S. Building Microservices: Designing Fine-Grained Systems [Text] / Sam Newman. — Sebastopol : O'Reilly Media, 2015. — 280 p.
13. Cherny, B. Programming TypeScript: Making Your JavaScript Applications Scale [Text] / Boris Cherny. — Sebastopol : O'Reilly Media, 2019. — 324 p.
14. Protic, M. How To Setup Rollup Config [Електронний ресурс] / Milos Protic. — Режим доступу: https://dev.to/proticm/how-to-setup-rollup-config-45mk. — Дата доступу: 27.09.2019.
15. Davidson, M. rollup-plugin-sourcemaps [Електронний ресурс] / Max Davidson. — Режим доступу: https://github.com/maxdavidson/rollup-plugin-sourcemaps. — Дата доступу: 05.10.2019.
16. Harris, R. rollup-plugin-node-resolve [Електронний ресурс] / Rich Harris. — Режим доступу: https://github.com/rollup/rollup-plugin-node-resolve. — Дата доступу: 05.10.2019.
17. Zolenko, E. rollup-plugin-typescript2 [Електронний ресурс] / Eugene Zolenko. — Режим доступу: https://github.com/ezolenko/rollup-plugin-typescript2. — Дата доступу: 05.10.2019.
18. Bardadym, D. rollup-plugin-visualizer [Електронний ресурс] / Denis Badradym. — Режим доступу: https://github.com/btd/rollup-plugin-visualizer. — Дата доступу: 05.10.2019.
19. Chadkin, B. rollup-plugin-terser [Електронний ресурс] / Bogdan Chadkin. — Режим доступу: https://github.com/TrySound/rollup-plugin-terser. — Дата доступу: 05.10.2019.

20. Strobel, M. rollup-plugin-gzip [Електронний ресурс] / Michael Strobel. — Режим доступу: https://github.com/kryops/rollup-plugin-gzip. — Дата доступу: 05.10.2019.
21. Hauser, E. apollo-link-http [Електронний ресурс] / Evans Hauser. — Режим доступу: https://github.com/apollographql/apollo-link/tree/master/packages/apollo-link-http. — Дата доступу: 10.10.2019.
22. Hauser, E. apollo-link-ws [Електронний ресурс] / Evans Hauser. — Режим доступу: https://github.com/apollographql/apollo-link/tree/master/packages/apollo-link-ws. — Дата доступу: 10.10.2019.
23. Patton, R. Software Testing [Text] / Ron Patton. — 2nd edition. — Carmel : Sams Publishing, 2005. — 408 p.
24. Lombart, T. How to test JavaScript with Jest [Електронний ресурс] / Thomas Lombart. — Режим доступу: https://dev.to/thomlom/how-to-test-javascript-with-jest-3bkg. — Дата доступу: 05.11.2019.
25. Palantir Technologies, TSLint [Електронний ресурс] / Palantir Technologies. — Режим доступу: https://palantir.github.io/tslint/. — Дата доступу: 10.11.2019.
26. Trostler, M. Testable JavaScript [Text] / Mark Ethan Trostler. — Sebastopol : O'Reilly Media, 2013. — 274 p.
27. Fraile, R. How GZIP compression works [Електронний ресурс] / Raul Fraile. — Режим доступу: https://2014.jsconf.eu/speakers/raul-fraile-how-gzip-compression-works.html. — Дата доступу: 12.11.2019.
28. Apollo, Apollo Link [Електронний ресурс] / Apollo. — Режим доступу: https://github.com/apollographql/apollo-link. — Дата доступу: 15.11.2019.
29. Hunt, M. How to invalidate cached data in Apollo and handle updating paginated queries [Електронний ресурс] / Martin Hunt. — Режим доступу: https://medium.com/@martinseanhunt/how-to-invalidate-cached-data-in-apollo-and-handle-updating-paginated-queries-379e4b9e4698. — Дата доступу: 19.11.2019.

ДОДАТКИ

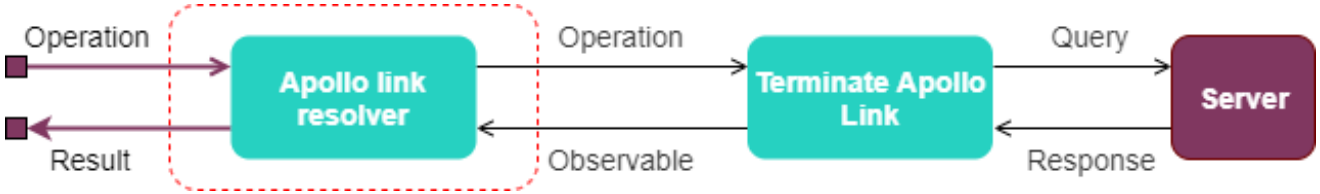
Додаток 1

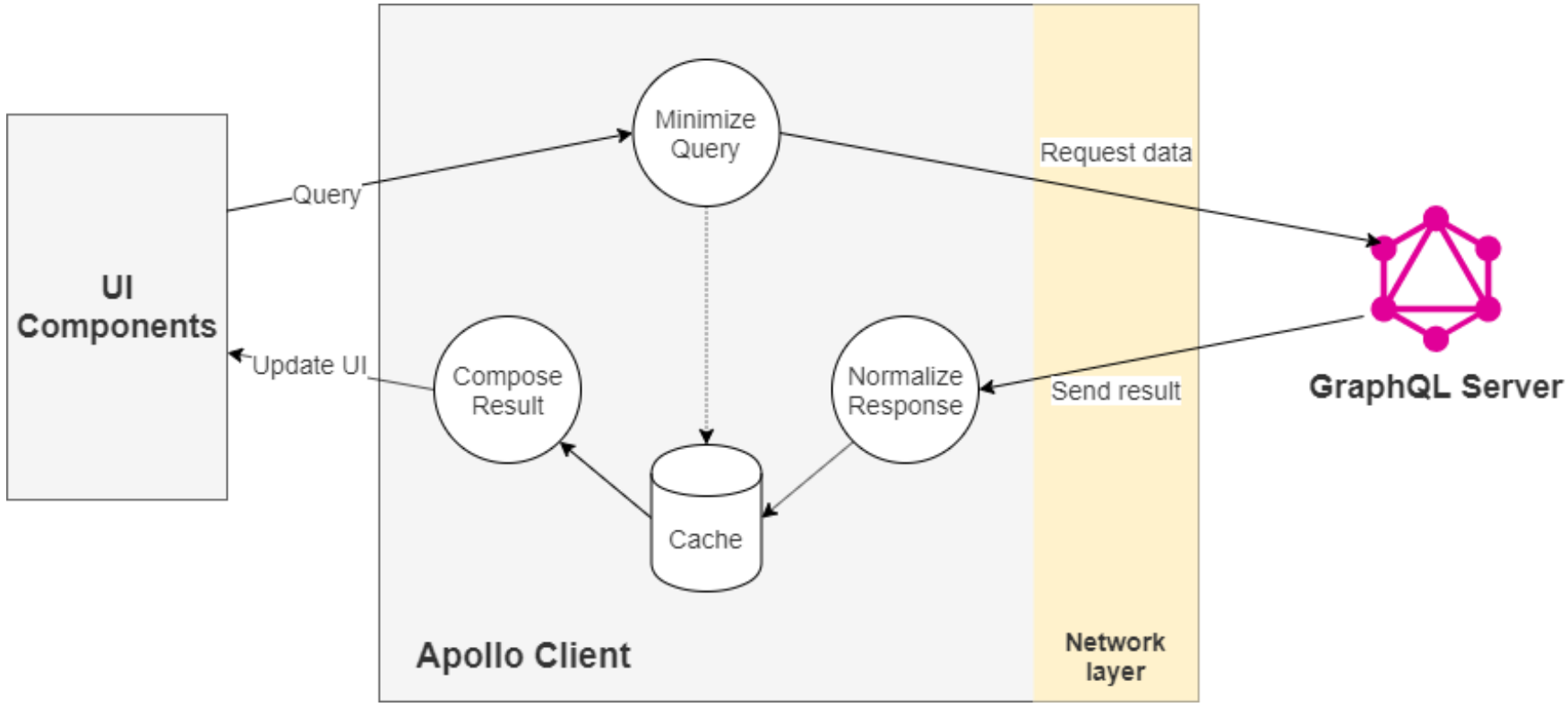
Копії графічних матеріалів

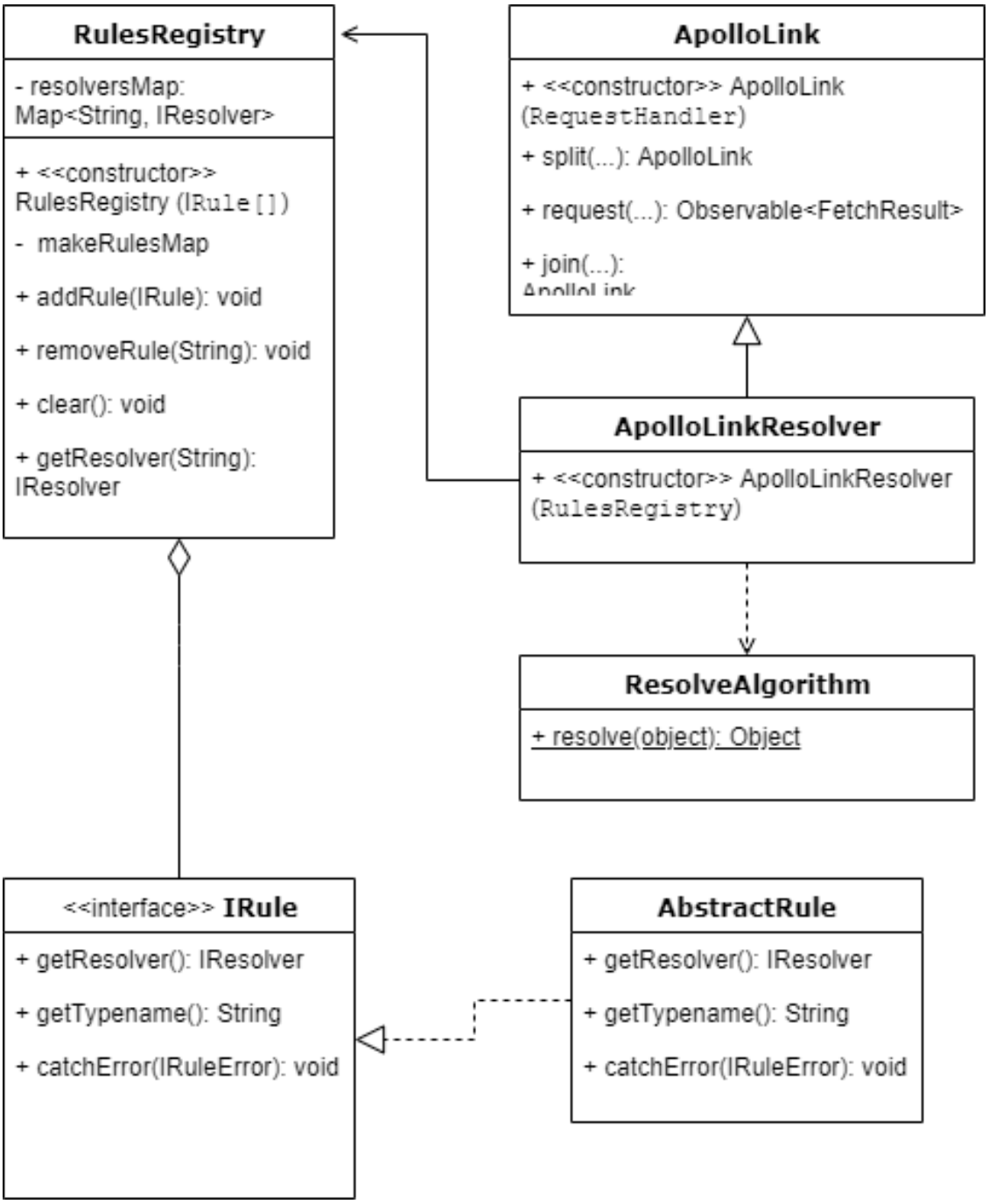

GraphQL Client
GraphQL Service
User Interface Components
Query
Minimize Query
Query
Tokenize
Tokens
Parse
Abstract Syntax Tree (AST)
Update UI
Compose Results
Validate
Schema
Error Response
Update Cache
Rewrite, Analyze
Analyzers
Simplified Tree
Normalize
Response
Execute
Resolvers
Call API endpoints, databases, etc.

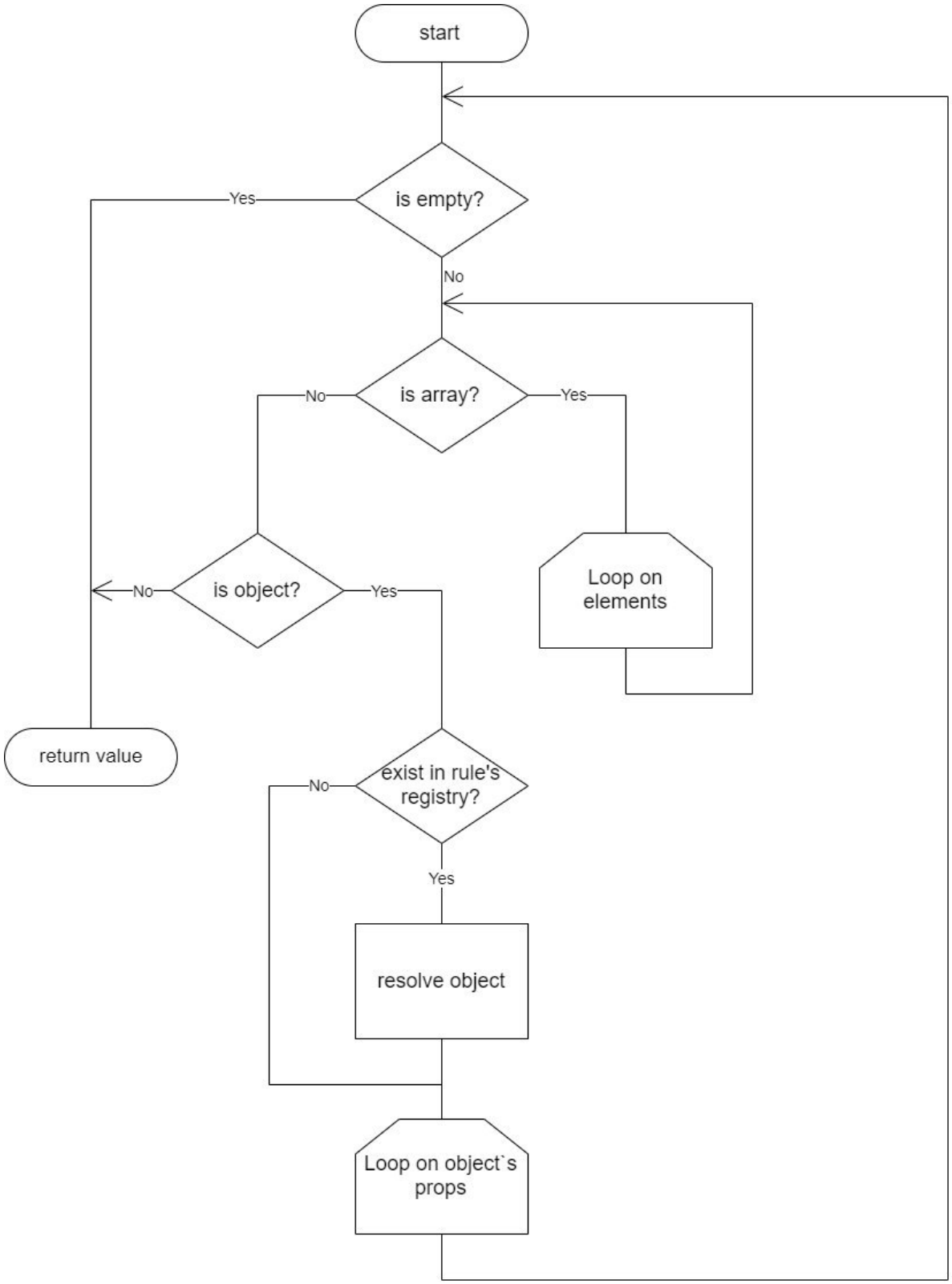

OPERATION
LINK
OBSERVABLE

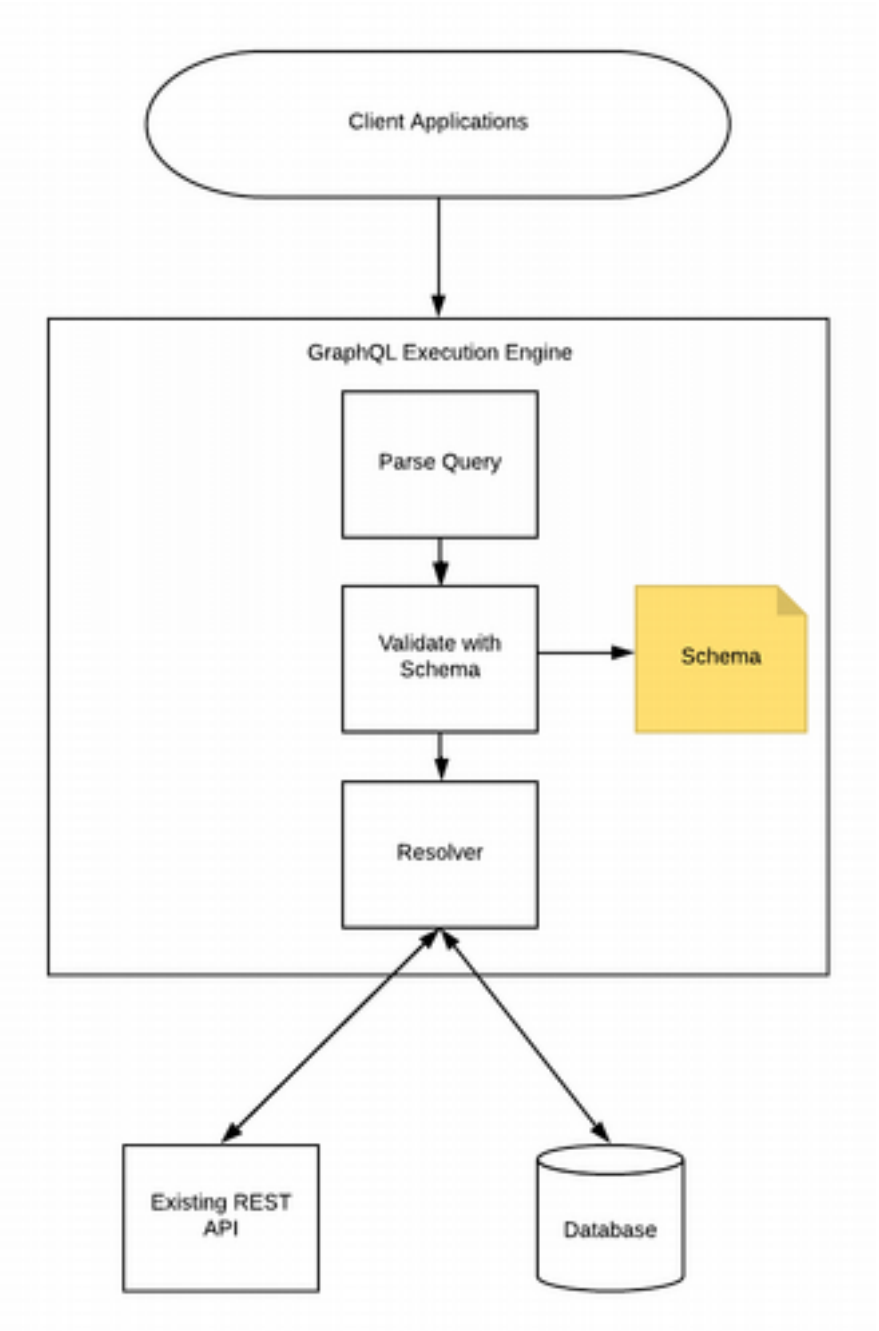

OPERATION
LINK
LINK
LINK
SERVER
RESULT


Operation
Apollo link resolver
Operation
Terminate Apollo Link
Query
Server
Result
Observable
Response


UI
Components
Query
Update UI
Minimize
Query
Compose
Result
Normalize
Response
Cache
Apollo Client
Request data
Send result
Network
layer
GraphQL Server


RulesRegistry
- resolversMap: Map<String, IResolver>
+ <<constructor>> RulesRegistry (IRule[])
- makeRulesMap
+ addRule(IRule): void
+ removeRule(String): void
+ clear(): void
+ getResolver(String): IResolver
ApolloLink
+ <<constructor>> ApolloLink (RequestHandler)
+ split(...): ApolloLink
+ request(...): Observable<FetchResult>
+ join(...):
ApolloLink
ApolloLinkResolver
+ <<constructor>> ApolloLinkResolver (RulesRegistry)
ResolveAlgorithm
+ resolve(object): Object
<<interface>> IRule
+ getResolver(): IResolver
+ getTypename(): String
+ catchError(IRuleError): void
AbstractRule
+ getResolver(): IResolver
+ getTypename(): String
+ catchError(IRuleError): void


start
is empty?
Yes
No
is array?
No
Yes
Loop on elements
is object?
No
Yes
return value
exist in rule's registry?
No
Yes
resolve object
Loop on object`s props


Client Applications
GraphQL Execution Engine
Parse Query
Validate with Schema
Schema
Resolver
Existing REST API
Database

Додаток 2

Копія презентації

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП’ЮТЕРНИХ СИСТЕМ

МОДИФІКОВАНИЙ СПОСІБ КЕШУВАННЯ ДАНИХ КЛІЄНТСЬКОЇ БІБЛІОТЕКИ APOLLO-CLIENT ДЛЯ GRAPHQL

Виконав: Худер Карім Нідаль

Науковий керівник: к.т.н. Люшенко Леся Анатоліївна

Київ – 2019

АКТУАЛЬНІСТЬ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Apollo Client на сьогоднішній день є найпопулярнішим клієнтом для роботи з GraphQL. В ході аналізу цієї бібліотеки було виявлено критичні помилки при роботі з контекстно-залежними сутностями.

Враховуючи популярність бібліотеки, вирішення даної проблеми є актуальним завданням.

Об’єкт дослідження: механізм кешування даних в бібліотеці Apollo Client.

Предмет дослідження: алгоритми та методи кешування даних

Наукове завдання: проаналізувати існуючий механізм кешування даних в бібліотеці Apollo Client, та запропонувати ефективний спосіб збереження контекстно-залежних сутностей.

Мета дослідження: розробка та дослідження ефективності програмного забезпечення, яке вирішує проблему кешування контекстно-залежних сутностей у бібліотеці Apollo Client, не змінюючи її API

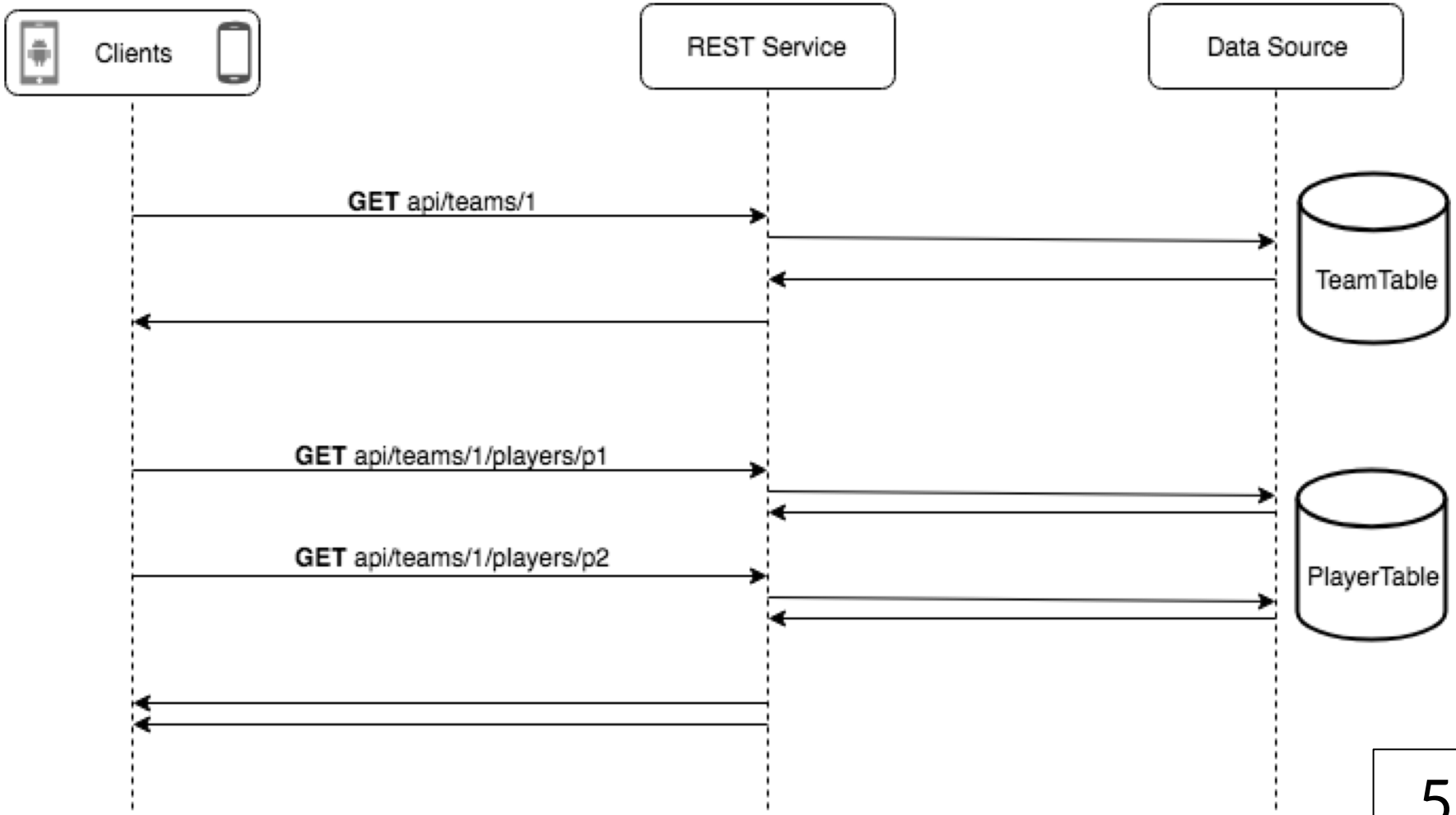
Окремі завдання

- дослідження та аналіз існуючого механізму кешування даних в бібліотеці Apollo Client
- розробка модифікованого способу кешування даних в бібліотеці Apollo Client, який дозволить реалізовувати:
- створення власних правил обробки даних;
- розробку Apollo Link Resolver, що оброблює результати GraphQL запитів з мережі;
- розробку алгоритму модифікації даних на основі, вказаних розробником, правил;
- підтримку усіх JavaScript середовищ;

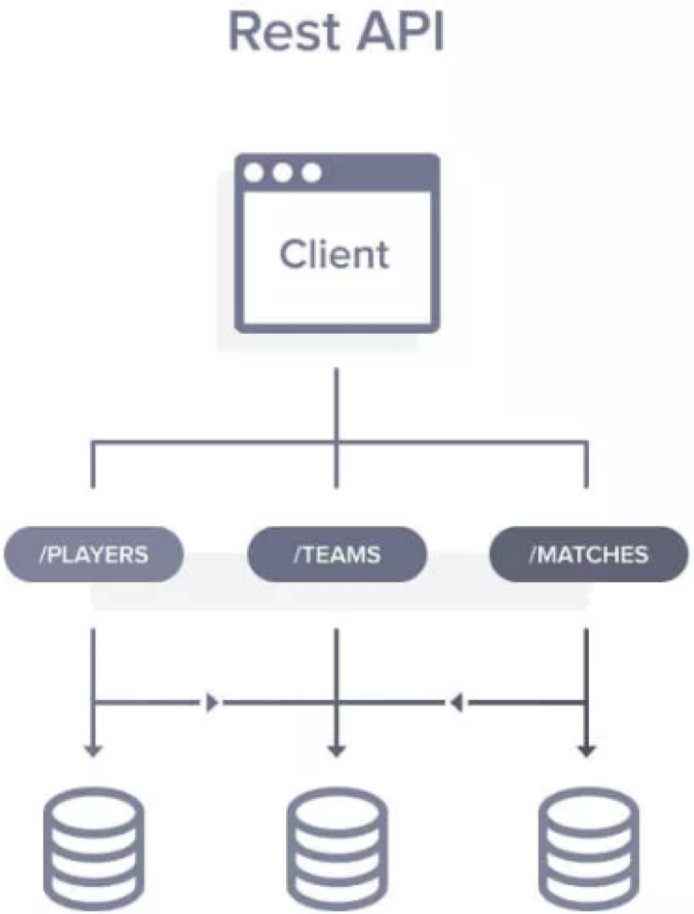
ОСНОВНІ ПРОБЛЕМИ REST

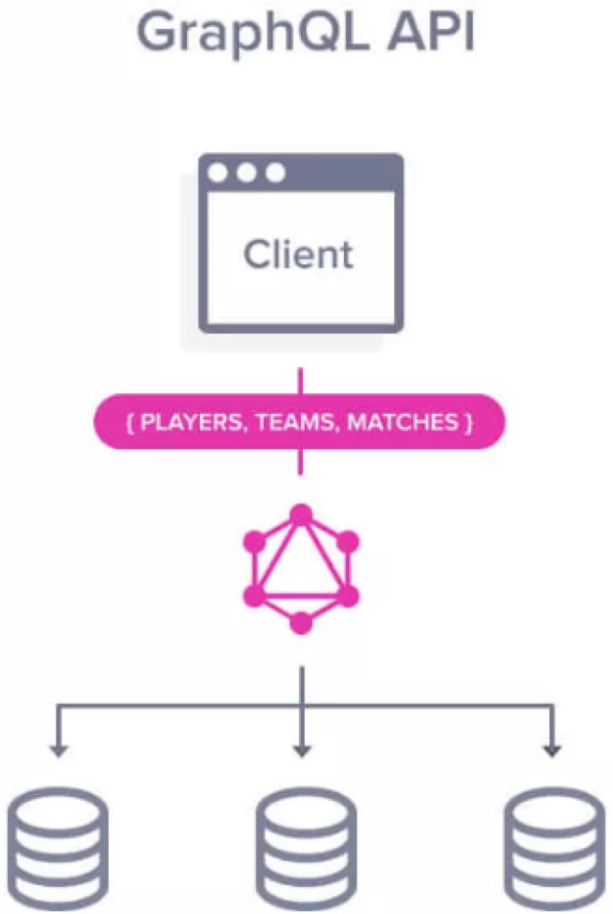
- Перезавантаження даних (Overfetching) — клієнт завантажує більше інформації, ніж потрібно в додатку;
- Недозавантаження даних (Underfetching) — конкретна кінцева точка не дає достатньої кількості необхідної інформації.
- Клієнту доведеться робити додаткові запити, щоб отримати всю необхідну структури даних.

ГРАФІЧНЕ ЗОБРАЖЕННЯ ПРОБЛЕМ REST



ПОРІВНЯННЯ КЛІЄНТ-СЕРВЕРНОЇ ВЗАЄМОДІЇ REST ТА GRAPHQL

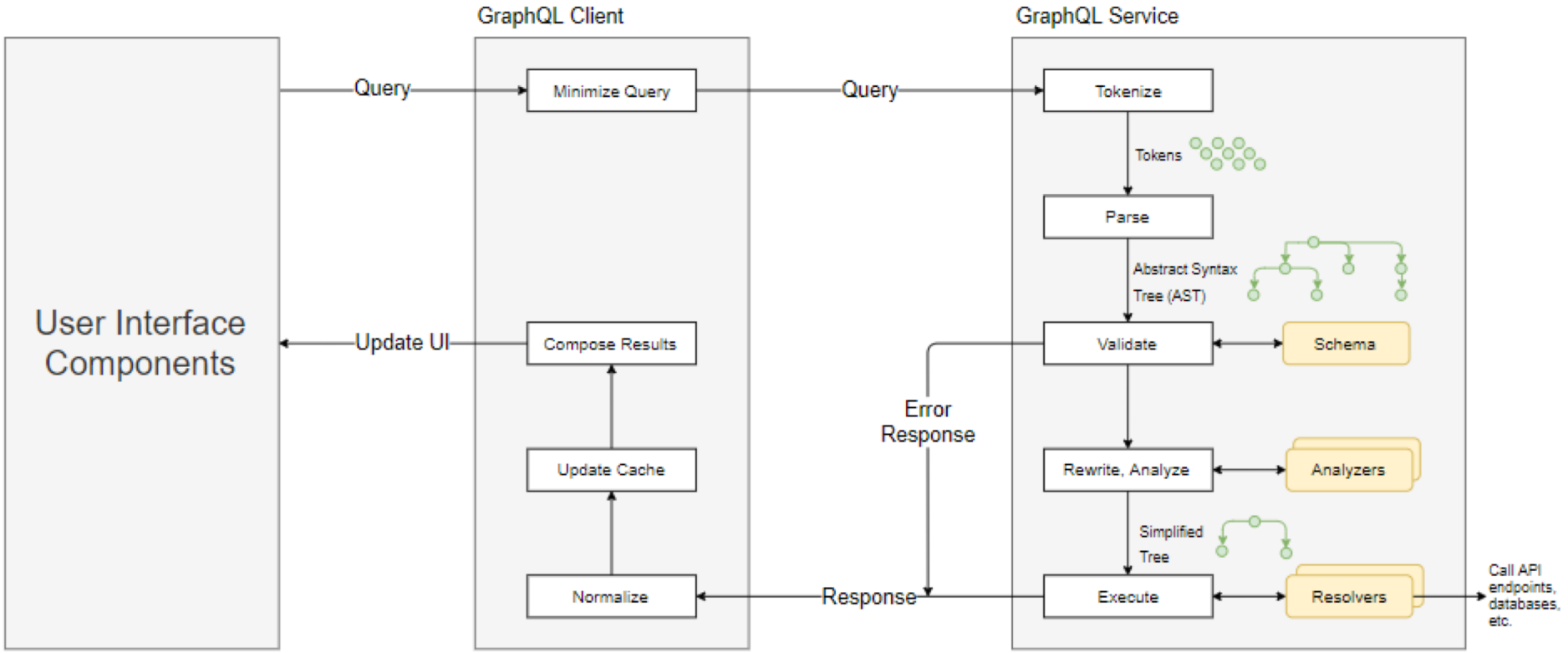




ПЕРЕВАГИ GRAPHQL

- єдина точка взаємодії з клієнтом;
- вирішена проблема перезавантаження та недостатнього завантаження даних;
- формування даних в залежності від клієнтського запиту;
- валідація даних;
- незалежність від транспортного рівня;
- автоматичне генерування документації для API.

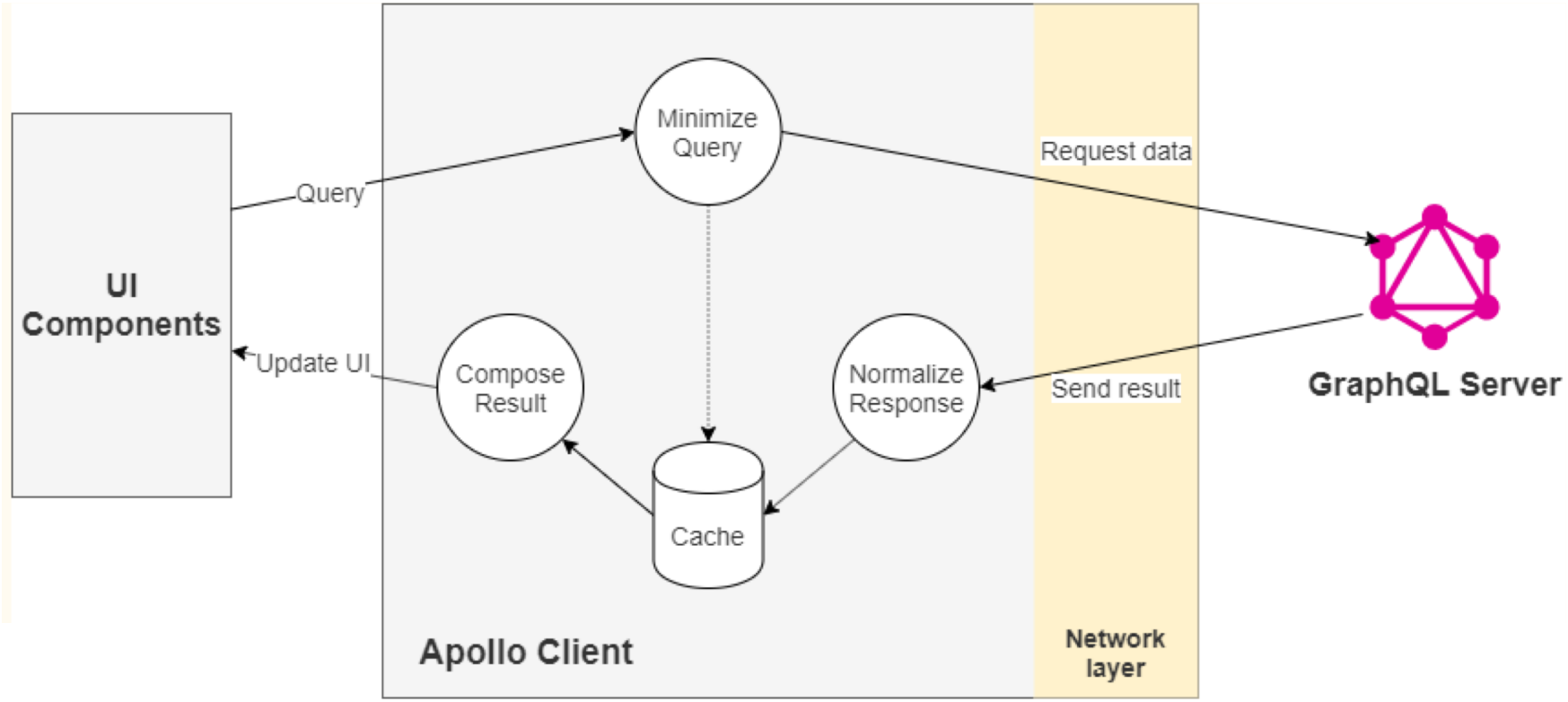
АРХІТЕКТУРА GRAPQHL



ФУНКЦІЇ GRAPHQL CLIENT

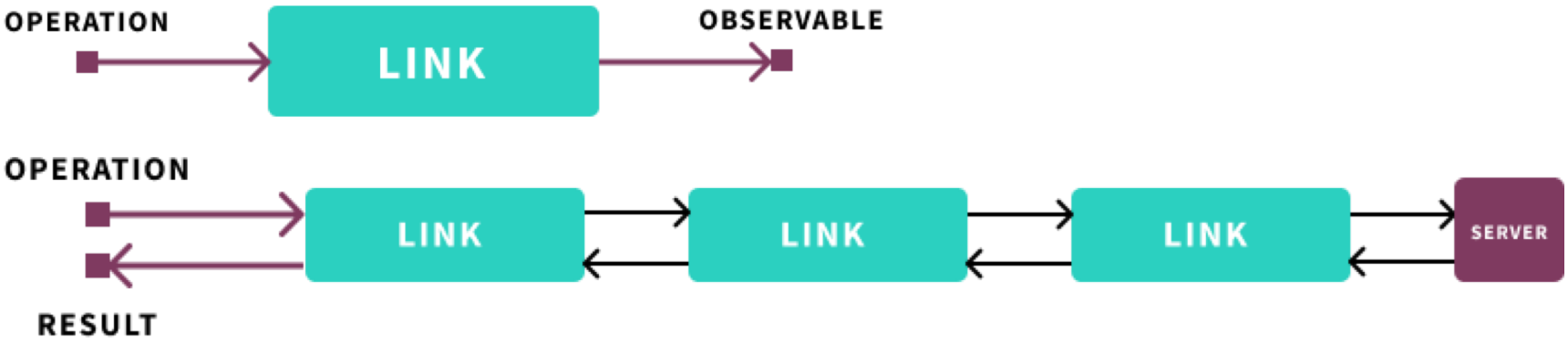
- мініфікація запитів;
- відправка GraphQL запитів на сервер;
- інтеграція з компонентами інтерфейсу;
- кешування даних;
- валідація та оптимізація GraphQL запитів на основі схеми.

АРХІТЕКТУРА APOLLO CLIENT



МЕРЕЖЕВИЙ ШАР APOLLO CLIENT

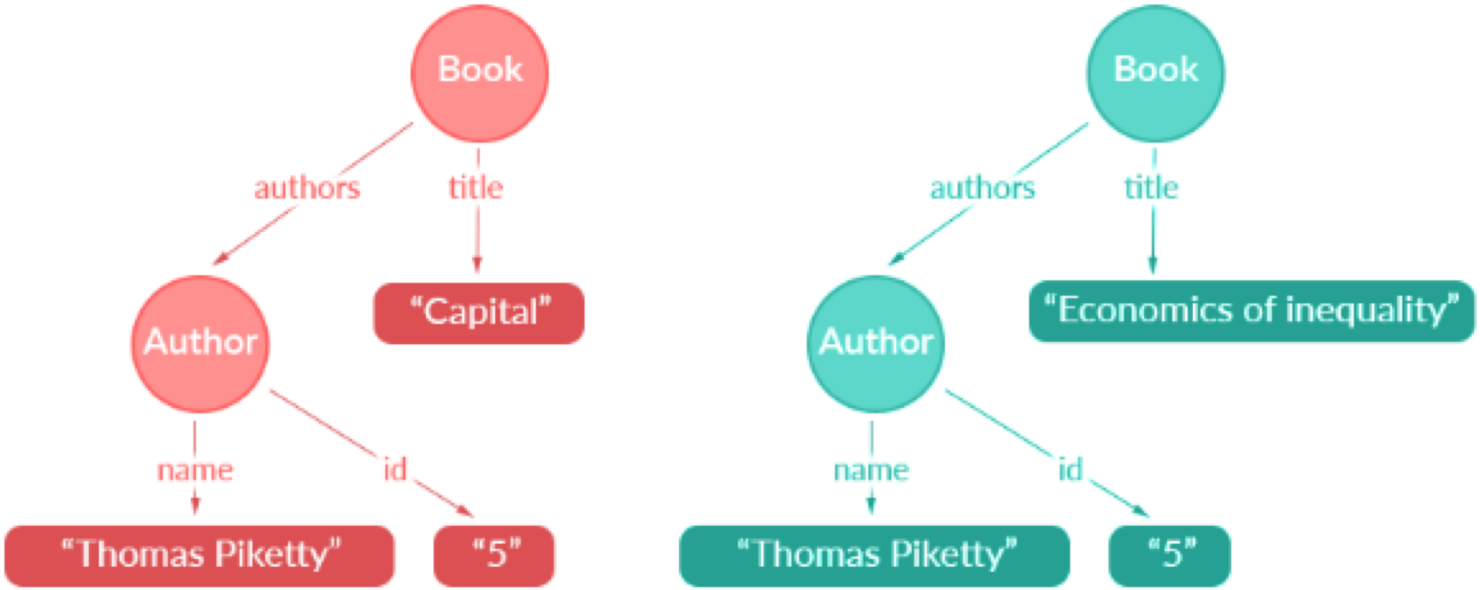
Основною складовою мережевого шару бібліотеки Apollo Client є Apollo Link.
Мережевий шар формується шляхом об’єднання декількох Apollo Link в єдиний ланцюг.

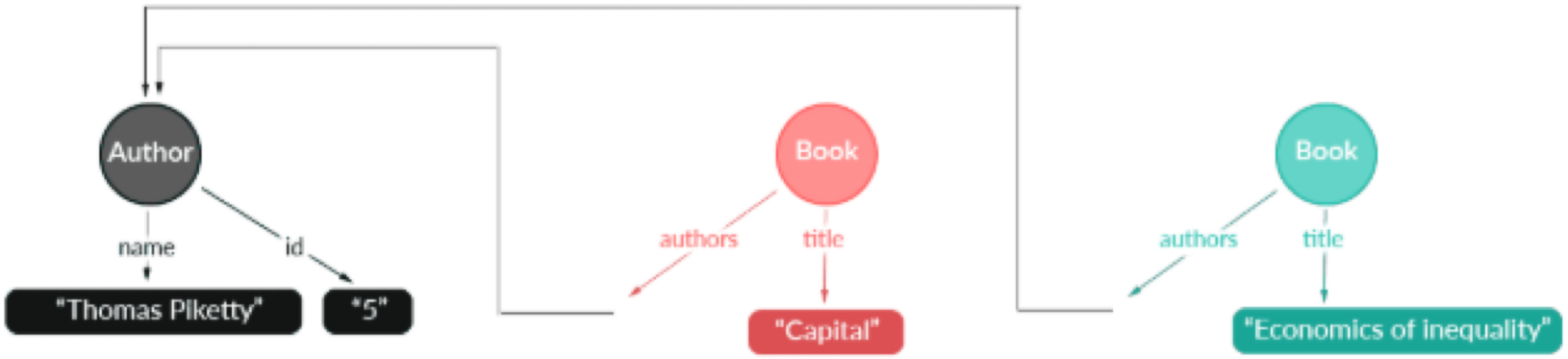


ПРОЦЕС КЕШУВАННЯ ДАНИХ

- розбиття результатів на окремі об’єкти;
- присвоєння унікальних ідентифікаторів для кожного об’єкту;
- валідація даних зі схемою;
- збереження даних у пласкому вигляді.

ГРАФІЧНЕ ПРЕДСТАВЛЕННЯ КЕШУВАННЯ ДАННИХ





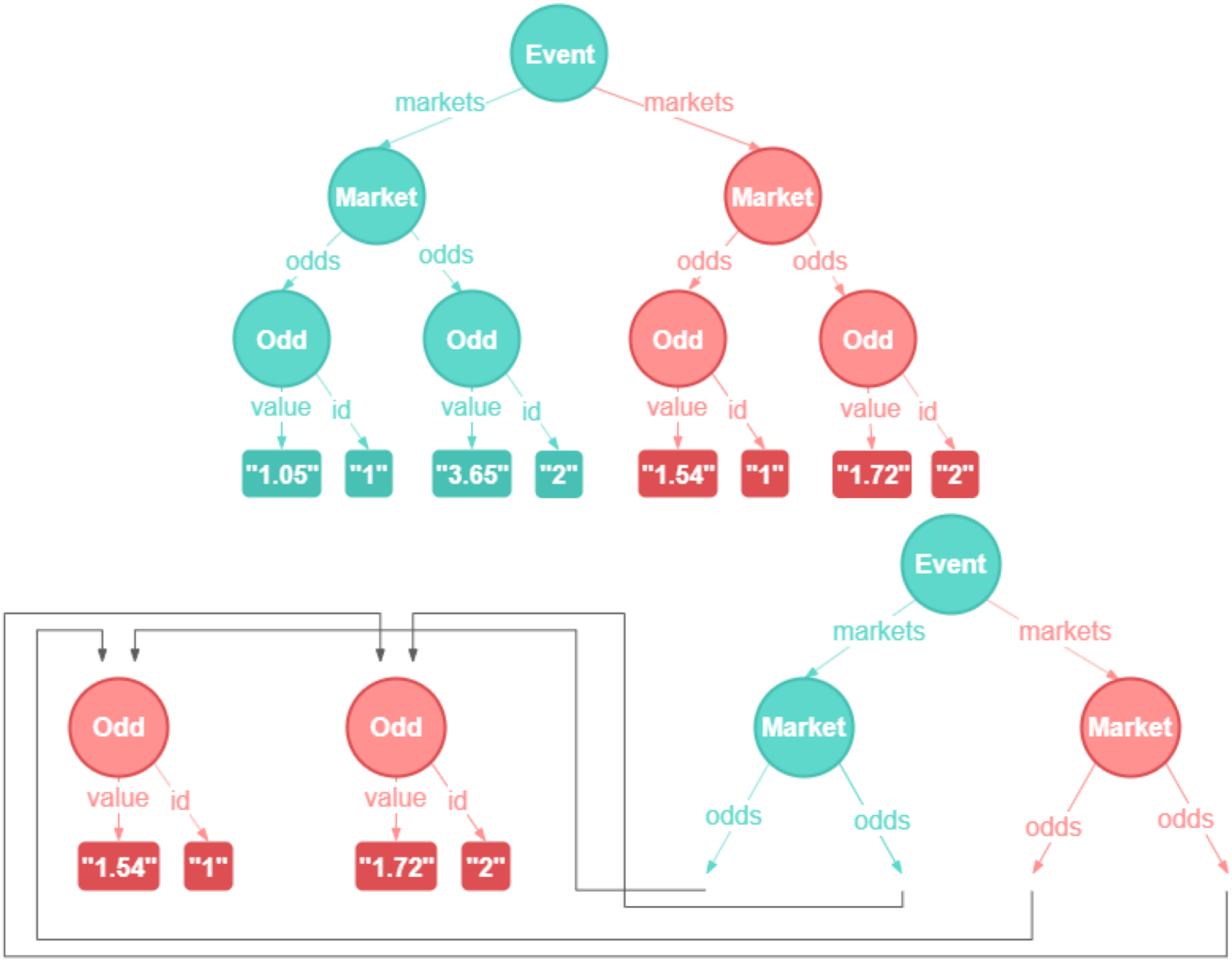
ПРОБЛЕМА КЕШУВАННЯ

Було виявлено проблему кешування контекстно-залежних сутностей.

Проблема:

- ідентифікатор сутності є унікальним тільки в рамках конкретного контексту.
- Apollo Client приймає ідентифікатор об’єкту ключем для кешування. В результаті дані перезаписуються.

ГРАФІЧНЕ ЗОБРАЖЕННЯ ПРОБЛЕМИ



ПРОБЛЕМИ ПРИ РОЗРОБЦІ РІШЕННЯ

Основні проблеми при розробці рішення для вирішення проблеми збереження контекстно-залежних сутностей у бібліотеці Apollo Client:

- Збереження зовнішнього API бібліотеки. Зміна API призведе до необхідності зміни існуючої кодової бази;
- Незалежність від конкретної схеми даних.

МОДИФІКОВАНИЙ СПОСІБ КЕШУВАННЯ ДАНИХ

1. Перехват результату запиту з мережі до моменту кешування.
2. Розбиття результату на окремі об’єкти.
3. Модифікація об’єктів за вказаними розробником правилами. Генерування композитних ідентифікаторів.
4. Передача модифікованого результату до стандартного механізму кешування Apollo Client

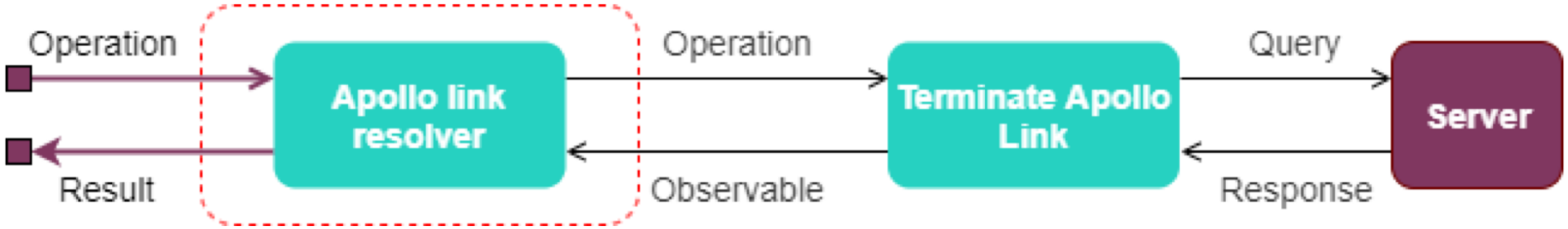
ПРАВИЛА ОБРОБКИ СУТНОСТЕЙ

Правило обробки сутностей — абстракції над процесом модифікації об’єкту конкретного типу. Дозволяє гнучко налаштовувати процес модифікації результату запиту для різних схем даних.

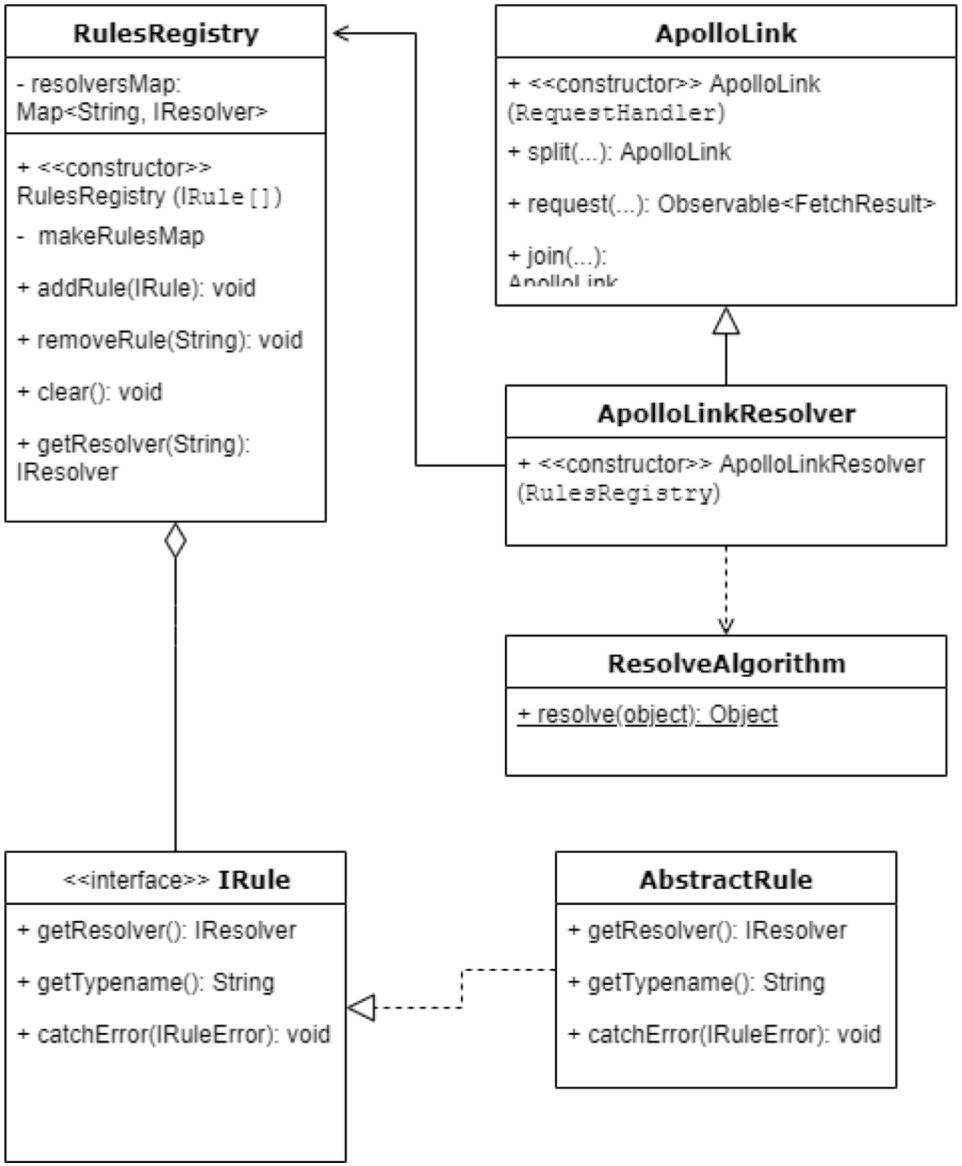
Інформація, яку містить правило:

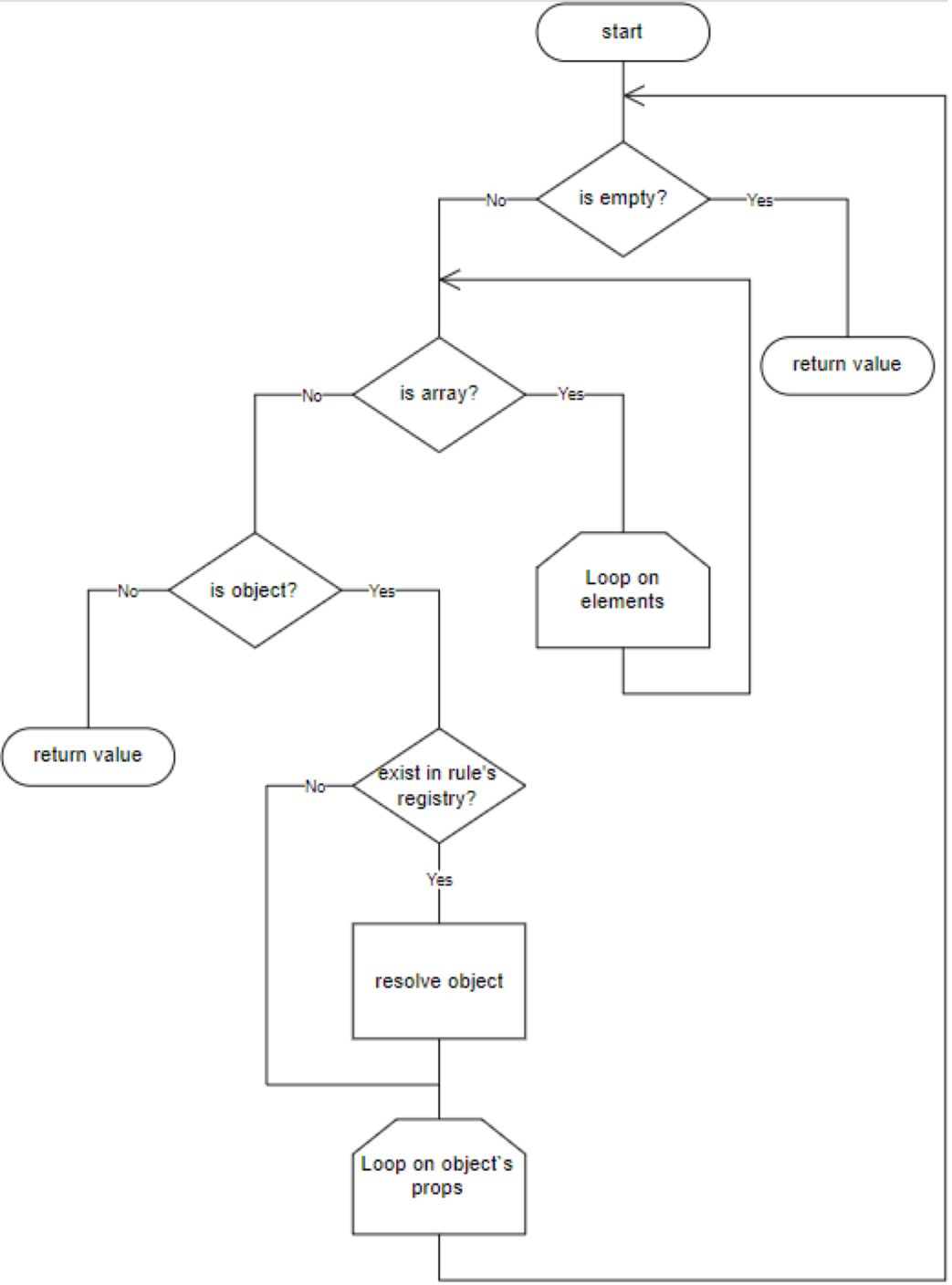
- Тип об’єкту для якого можна використовувати дане правило.
- Функція обробки помилки.
- Функція модифікації.

ГРАФІЧНЕ ПРЕДСТАВЛЕННЯ РІШЕННЯ



ДІАГРАМА КЛАСІВ РОЗРОБЛЕНОЇ БІБЛІОТЕКИ





АЛГОРИТМ ОБРОБКИ ДАНИХ


Національний технічний університет України
"Київський політехнічний інститут"

ТЕХНОЛОГІЇ


APOLLO


TS


GraphQL


TS
Lint


Jest


rollup.js

ТЕСТУВАННЯ

All files

100% Statements 75/75 **100%** Branches 20/20 **100%** Functions 19/19 **100%** Lines 66/66

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File	Statements		Branches		Functions		Lines	
apollo-link-resolver	100%	19/19	100%	4/4	100%	5/5	100%	18/18
resolve	100%	20/20	100%	10/10	100%	4/4	100%	17/17
rule	100%	36/36	100%	6/6	100%	10/10	100%	31/31

File	Statements		Branches		Functions		Lines	
apollo-link-resolver.ts	100%	17/17	100%	4/4	100%	5/5	100%	17/17
index.ts	100%	2/2	100%	0/0	100%	0/0	100%	1/1

File	Statements		Branches		Functions		Lines	
index.ts	100%	2/2	100%	0/0	100%	0/0	100%	1/1
resolve.ts	100%	18/18	100%	10/10	100%	4/4	100%	16/16

File	Statements		Branches		Functions		Lines	
index.ts	100%	4/4	100%	0/0	100%	0/0	100%	2/2
registry.ts	100%	23/23	100%	6/6	100%	8/8	100%	21/21
rule.ts	100%	9/9	100%	0/0	100%	2/2	100%	8/8

ФІНАНСОВИЙ ПЛАН СТАРТАПУ

Вид витрат	Вартість ($)
Юридичні послуги	350
Комп’ютерна техніка та ПЗ	2000
Засоби зв’язку	50
Заробітна плата	3000
Податок	300

Тривалість проекту: 1 місяць.
Кінцева вартість проекту: 5 700.
Кінцева ціна продукту: 20 000.
Прибуток: 14 300.

ФІНАНСОВИЙ ПЛАН СТАРТАПУ

Вид витрат	Вартість ($)
Стартові	
Юридичні послуги	350
Комп'ютерна техніка та ПЗ	3000
Ріелторські послуги, меблі та канцелярія	2100
Транспортні витрати	150
Щомісячні	
Зарплата за посадами	3500
Оренда офісного приміщення	600
Абонентська плата за засоби зв'язку	50
Податки	300
Непередбачені витрати	300

Сума для старту проекту: 5 600
Щомісячні витрати: 4 750
Вартість ліцензії: 2

ВИМОГИ ДО ВИКОРИСТАННЯ РОЗРОБЛЕНОЇ БІБЛІОТЕКИ

1. Apollo Client v2
2. Правила не повинні змінювати схему даних
3. В проекті має бути встановлена залежність Apollo Link

НАУКОВО-ІНОВАЦІЙНА НОВИЗНА

Розроблено модифікований спосіб кешування даних для бібліотеки Apollo Client, який вирішує проблему збереження та роботи з контекстно-залежними стуностями.

ВИСНОВКИ

1. Було досліджено архітектуру бібліотеки Apollo Client та усіх її складових.
2. Було досліджено проблему коректної роботи кешу з контекстно-залежними сутностями.
3. Було розроблено бібліотеку, яка надає інструменти для вирішення проблеми збереження контекстно залежних сутностей:
   - Apollo Link Resolver
   - Rule
   - Rule Registry

Унікальність

1.32% Matches

Highest match: **0.48%** with library source. File ID: **8417387**

0.54% Internet Matches	30	Page 55
1.32% Library matches	96	Page 55

0% Quotes

No quotes found

0% Exclusions

No exclusions found

Replacement

No replaced characters found

АПРОБУВАННЯ ОТРИМАНИХ РЕЗУЛЬТАТІВ

1XII наукова конференція магістрантів та аспірантів «Прикладна математика та комп’ютинг» (ПМК-2019).

Дякую за увагу!